Extension paramétrée de compilateur certifié pour la programmation parallèle

Sylvain Dailler

Centre Val de Loire UNIVERSITÉ Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

THÈSE présentée par :

Sylvain DAILLER

soutenue le : 17 Décembre 2015

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline : **Informatique**

Extension paramétrée de compilateur certifié pour la programmation parallèle

THÈSE DIRIGÉE PAR :

Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

Gaétan HAINS	Professeur, Huawei
Ludovic HENRIO	Chargé de recherches, CNRS

JURY :

Jean-Michel COUVREUR	Professeur, Université d'Orléans, Président du jury
Pierre COURTIEU	Maître de Conférences, CNAM
Frédéric DABROWSKI	Maître de Conférences, Université d'Orléans

Table des matières

Liste des figures

1 Introduction

Sommaire

1.1 Contexte

Depuis l'apparition des premiers ordinateurs, les applications informatiques n'ont cessé de croître en quantité, en diversité et en complexité. De nos jours, on peut trouver des programmes informatiques partout. En particulier, ceux-ci sont de plus en plus présents dans les systèmes critiques qui interviennent dans différents domaines, comme par exemple :

— la médecine : chirurgie assistée par ordinateur, aide au diagnostique et vérification des contre-indications médicamenteuse, ...
— les transports : aide au pilotage, pilotage automatique (avions, voitures, métros, fusées...).
— l'analyse financière : prise de décisions, gestion d'informations confidentielles, ...

Pour de tels systèmes les conséquences d'une erreur ont un coût inacceptable, que ce soit sur le plan humain ou financier.

Afin de limiter la présence d'erreurs dans les systèmes informatiques, l'approche la plus répandue consiste à tester le programme sur certaines entrées. On distingue différentes techniques de tests qui peuvent être plus ou moins perfectionnées en fonction du niveau de criticité des systèmes considérés. Les jeux de tests (collections de cas d'utilisation du système) peuvent être écrits à la main ou générés automatiquement (par exemple en utilisant un générateur aléatoire). Dans tous les cas l'objectif est d'obtenir une couverture maximale des exécutions

possibles d'un programme. La génération de tests pertinents reste un problème complexe et un champ de recherche actif. Cette approche souffre toutefois d'une limitation inhérente à la nature des objets considérés : il n'est pas possible d'obtenir une couverture complète des situations possibles. La seule exception est celle des systèmes finis (le nombre de situations possibles est fini). Dans une très grande partie des cas, les tests nécessitent des ressources considérables rendant impossible l'examen de toutes les exécutions possibles. La correction d'une classe importante de programmes ne peut donc pas être assurée par des méthodes de tests.

La spécification formelle et l'analyse de programmes permettent de caractériser de manière exacte les propriétés (exprimées sous la forme de formules logiques) décrivant le comportement d'un programme. Contrairement aux méthodes précédentes, il n'est pas question ici de tester chaque cas d'exécution indépendemment pour vérifier qu'il satisfait une propriété donnée. L'analyse de programme repose sur l'observation du code source du programme et un raisonnement logique qui permettent de démontrer que toutes les exécutions possibles du programme vérifient la propriété attendue. Il existe plusieurs méthodes d'analyse formelle comme le typage, le *model checking*, l'interprétation abstraite ou la vérification déductive. Ces méthodes diffèrent les unes des autres par le niveau d'expressivité des propriétés de correction qu'elles permettent d'exprimer, le degré d'automatisation possible, *etc.* Bien que ces méthodes soient généralement complexes à mettre en place (grandes consommatrices de ressources humaines et de puissance de calcul, niveau élevé de formation des intervenants, *etc.*), on constate que ces méthodes sont de plus en plus utilisées. En particulier, on peut noter les exemples suivants impliquant des industriels :

— le logiciel embarqué utilisé dans le métro parisien est prouvé à l'aide de la méthode B,
— le logiciel d'interprétation abstraite ASTREE est utilisé dans l'aéronautique pour assurer l'absence d'une classe d'erreur à l'exécution des logiciels embarqués.

Cette généralisation de l'utilisation des méthodes formelles dans le cadre du développement logiciel s'explique principalement par le niveau de confiance offert par celles-ci et, dans une moindre mesure, par le fait que l'investissement nécessaire en amont est en partie compensé par les gains obtenus en terme de temps de debugging (pour un niveau d'exigence suffisant).

Le langage (langage machine, également appelé assembleur) dans lequel sont exprimés les programmes s'exécutant sur une machine est extrêmement complexe, écrire des programmes pour ce langage est très difficile et source d'erreurs : en général, on n'en écrit directement que pour optimiser le temps d'exécution de certains programmes. En pratique, la majorité des programmes s'écrit donc dans un langage de haut-niveau avant d'être compilé (traduit automatiquement) en du langage machine. De la même manière, il est difficile d'appliquer les techniques de méthodes formelles directement sur le langage machine et on préfère réaliser le travail de vérification dans le langage de haut-niveau. Or, c'est le langage machine qui sera finalement exécuté ; toute erreur de traduction au cours de la compilation invalide donc les efforts de formalisation réalisés en amont. Une solution à ce problème consiste à appliquer les mêmes techniques à la preuve de correction du compilateur afin de s'assurer de la conservation des propriétés des programmes au cours de la traduction. Concrètement,

— on prouve que le compilateur est correct ; le programme obtenu en langage machine aura le même comportement que le programme source,
— on applique nos techniques de méthodes formelles sur un programme écrit dans un langage de haut-niveau afin de montrer que son comportement possède certaines propriétés,
— on combine ces deux résultats pour s'assurer que le comportement du programme qui sera effectivement exécuté possède également ces propriétés.

Les travaux présentés par la suite portent sur la preuve de correction de compilateurs. Parce qu'un compilateur est un objet complexe, la preuve de sa correction ne peut se faire qu'au travers de l'utilisation d'outils très rigoureux. On pense en particulier à la vérification déductive de programmes qui est utilisée ici. En pratique, avec cette méthode, il est difficile de prouver manuellement qu'un programme suit une spécification donnée car la taille des démonstrations est considérable (de l'ordre de dizaines de milliers de lignes). Le recours à l'utilisation d'un assistant de preuve est dans ce cas indispensable ; ce dernier apporte de plus un niveau de confiance extrêmement élevée. Intuitivement, on peut donner les règles de raisonnement que l'on va utiliser à une machine en vue de lui faire vérifier les étapes de notre raisonnement. Le logiciel permettant cela (appelé un *proof checker*) ne fait théoriquement pas d'erreurs et peut vérifier une quantité très importante de résultats rapidement. Le théorème des 4 couleurs, par exemple, a été démontré en utilisant l'assistant de preuve Coq alors qu'aucune démons-

tration « manuelle » du théorème n'existe (la preuve repose sur une disjonction de cas si importante qu'elle n'est pas vérifiable par un humain). Un *proof checker* permet la vérification mécanique de démonstrations écrites dans un langage qui lui est propre. Cet outil est en général une partie d'un assistant de preuve qui permet d'aider à écrire des démonstrations dans ce langage et fournit souvent des outils d'automatisation pour une écriture plus rapide des preuves. Il existe beaucoup d'assistants de preuves parmi lesquels : Mizar, HOL-Isabelle, Coq, *etc.* On peut les différencier par le choix des axiomes sur lesquels ils reposent et leur niveau d'automatisation. Dans la suite, on parlera uniquement de Coq que nous avons choisi pour nos développements : il présente plusieurs avantages tels qu'une approche constructive, un langage de tactiques Ltac, et une communauté active (plusieurs résultats mathématiques importants ont été démontrés en Coq).

La preuve de correction mécanique de compilateur est une pièce essentielle de la validation formelle de programmes. Dans le cadre de systèmes critiques, le compilateur lui-même doit en effet être considéré comme un composant critique car de sa correction dépend la conservation des propriétés de correction des programmes traduits. Ces dernières années en particulier, un grand nombre de compilateurs certifiés ont été développés pour différents langages et différentes architectures. Ces compilateurs concernaient des langages et/ou architectures simplifiés et constituent avant tout des preuves de concept ; ce qui n'enlève rien à l'importance de leur contribution au domaine. Une avancée significative a été réalisée par le développement de Compcert [31] (par Leroy et al). Compcert est un compilateur certifié en Coq pour un sous-ensemble de C. Ce compilateur a ensuite été étendu à plusieurs architectures et sa spécification a évolué : par exemple, il ne garantissait à l'origine que la correction des programmes qui terminent. On ne pouvait donc pas parler de la correction de la compilation d'un programme qui ne termine pas forcément (par exemple un système d'exploitation). Le passage à l'échelle réalisé par ces travaux permet maintenant d'envisager l'utilisation de compilateurs certifiés dans le processus de validation de programmes séquentiels.

Au cours des dernières années, toutefois, la progression de la fréquence de l'horloge des microprocesseurs a été stoppée pour des raisons physiques (dissipation thermique, effets quantiques) poussant ainsi les fabricants à développer des architectures dîtes multi-cœurs. Ces nouveaux systèmes sont constitués de plusieurs processeurs fonctionnant simultanément et se partageant une mé-

moire. Dans le domaine de la compilation, ce partage de mémoire pose problème car les processeurs et compilateurs actuels font de nombreuses optimisations qui peuvent se traduire par des réordonnancements d'actions sur la mémoire (par exemple : un processeur va pouvoir exécuter une lecture avant une écriture sous certaines conditions). Ces optimisations sont prévues pour ne poser aucun problème dans le cadre d'un processeur mono-cœur. En revanche l'introduction d'accès concurrents à la mémoire peut conduire à des comportements inattendus. En particulier, on perd la consistance séquentielle, qui peut s'énoncer de la manière suivante en citant Lamport [29] :

> quelle que soit l'exécution considérée, le résultat est celui qui aurait été obtenu si les opérations de chaque processeur avaient été exécutées dans un ordre séquentiel donné qui préserve l'ordre des instructions de chaque processeur tel que spécifié par le programme.

Intuitivement, le programme induit un ordre sur les instructions exécutées par chaque processeur. Si, pour chaque processeur, cet ordre est respecté par toutes les exécutions possibles, on a la consistance séquentielle. En pratique, le comportement des architectures multi-cœurs repose sur des modèles mémoires dits faibles ou relâchés n'assurant pas la consistance séquentielle pour des raisons de performances. En fonction des architectures, le comportement des programmes peut varier selon des règles plus ou moins strictes. Par exemple, dans le modèle TSO (Total Store Ordering) [41] pour une architecture x86 le comportement suivant (exemple iwp2.3.a/amd4) est possible : soit deux emplacements mémoires x et y (initialisés à 0), si deux processeurs écrivent 1 dans x (respectivement y) puis lisent depuis y (respectivement x), il est possible qu'ils lisent 0 tous les deux. Cela s'explique par le fait que l'architecture autorise les deux processeurs à changer l'ordre des instructions de lecture et d'écriture.

La certification d'un compilateur pour une architecture en mémoire partagée est un problème difficile à cause du parallélisme et du modèle mémoire utilisé. L'équipe CompcertTSO a développé un compilateur d'un sous-ensemble de C sur la base de Compcert pour une architecture spécifique modélisée par le modèle TSO. Notre travail se positionne dans la continuité de ces travaux avec comme objectif la compilation certifiée de squelettes algorithmiques. Les squelettes algorithmiques sont un modèle de programmation parallèle haut-niveau. Afin de cacher les difficultés de la programmation parallèle au développeurs, ce modèle repose sur l'introduction de primitives séquentielles dont l'implémentation est parallélisée. L'objectif à long terme est le traitement de squelettes algorithmiques mais ce mémoire n'aborde qu'une première étape nécessaire à

leur compilation correcte. Ces squelettes ne seront donc plus évoqués dans ce mémoire.

1.2 Contribution

Notre contribution s'articule autour d'une extension sémantique modulaire (en Coq) d'un compilateur certifié existant Compcert et en particulier à sa version en mémoire partagée CompcertTSO. Nous pensons que les compilateurs Compcert et CompcertTSO souffrent de deux limitations qui ne nous permettent pas de les utiliser simplement pour la compilation de squelettes algorithmiques :

— La spécification est définie en fonction des traces d'exécution des programmes source et compilé. Informellement, la compilation est correcte si la trace d'exécution du programme cible est une des traces d'exécution possibles du programme source. Les éléments de cette trace sont des appels API/systèmes faits par le programme. Un premier problème est que l'on n'a aucun moyen de changer ses appels API/systèmes au cours de la compilation : on voudrait, par exemple, pouvoir traduire la création de n processus légers dans un squelette algorithmique vers n créations d'un processus léger dans le programme cible. Il est également impossible dans CompcertTSO d'influencer l'ordonnancement des processus légers autrement qu'avec des instructions atomiques de type *read-modify-write, compare-and-swap*. Il est cependant souhaitable de pouvoir raisonner avec des primitives de synchronisation plus haut-niveau. La solution que nous envisageons est d'ajouter un ordonnanceur sous la forme d'une bibliothèque de primitives parallèles dans la spécification du langage.

— Nous souhaitons être très modulaire. On voudrait pouvoir changer le modèle mémoire (dans la mesure du possible) sans réécrire tout le compilateur. Dans CompcertTSO, l'intégralité des langages intermédiaires repose sur le modèle TSO. Idéalement, la certification des phases de compilation indépendante du modèle mémoire considéré devraient pouvoir être réutilisées sans modification quel que soit le modèle mémoire de la cible du compilateur. Seules certaines phases d'optimisation devraient dépendre de la cible comme cela se passe dans le processus classique de développement d'un compilateur.

Notre objectif est de fournir une spécification d'un système n'incluant pas uniquement le programme dans un environnement clos mais d'y ajouter la possibilité d'instancier à la demande les éléments nécessaire au raisonnement dans un contexte parallèle (ordonnancement de processus légers et modèle mémoire). Nous appliquons donc les changements suivants au compilateur :

— Nous étendons la sémantique de CompcertTSO pour y ajouter une bibliothèque abstraite modulaire (ordonnanceur) : elle est définie par un jeu d'hypothèses et instanciable par un ordonnanceur sémantique pouvant contenir des *locks* entre autre. On fournit également une instance de la bibliothèque inspirée de la bibliothèque pthread. On a entre autre *locks*, *wait*, *join*.

— Nous redéfinissons le modèle mémoire de manière modulaire. L'idée étant d'obtenir un ensemble d'hypothèses modélisant un modèle mémoire abstrait compatible avec le modèle TSO mais également avec un modèle mémoire séquentiellement consistant. L'idée est que la spécification du langage source pourrait être faite sur un modèle séquentiellement consistant alors que celle du modèle cible est nécessairement TSO. On fournit également ces instances pour le modèle mémoire.

— Enfin, ces spécifications de modèles mémoires et d'ordonnanceur sont combinées dans une seule sémantique de système globale. On fournit une preuve de la correction du compilateur pour toute instance de ce système global abstrait en adaptant la preuve de compilation de CompcertTSO. Les instances nous permettent de retrouver la spécification de CompcertTSO en sachant que l'on a d'autres instances possibles.

1.3 Organisation du mémoire

Le mémoire s'organise en plusieurs chapitres :

— Le chapitre 2 concerne des rappels sur la vérification déductive (notamment l'assistant de preuves Coq), la compilation certifiée et les modèles mémoires faibles.

— Dans le chapitre 3, nous rappellerons l'état de l'art de ce travail. Nous reviendrons sur les travaux en compilation certifiée, sur les modèles mémoires ainsi que sur les travaux abordant la compilation certifiée de langage parallèle.

— Dans le chapitre 4, nous aborderons en détails les abstractions de modèles mémoires et de bibliothèques ainsi que les choix qui nous poussent à faire de telles abstractions. On expliquera nos choix de conception des langages abstraits et notamment l'ajout de machines mémoire, ordonnanceur, et globale (qui permet la synchronisation des autres machines).
— Dans le chapitre 5, nous montrerons qu'il est possible de se baser sur les preuves de correction du compilateur par processus léger, des différentes hypothèses que l'on place sur les machines mémoires et sur les machines ordonnanceurs pour obtenir un compilateur certifiée pour le langage étendu. On verra comment on peut obtenir une notion de *backward simulation* étendue puis que cette simulation implique la conservation de traces.
— Dans le chapitre 6, nous mettrons en relation les abstractions avec des propositions d'instances cohérentes. En particulier, on instancie l'abstraction du modèle mémoire par un modèle mémoire TSO puis par un modèle mémoire séquentiellement consistant. On démontre qu'ils sont compatibles (relation de simulation) sous certaines conditions. On crée aussi un ordonnanceur capable de gérer différentes primitives de synchronisation comme les primitives de type *locks* ou *wait, notify*.
— Nous conclurons dans le chapitre 7 et aborderons les perspectives de ce travail.

Notions préliminaires

2

Sommaire

Mes contributions concernent la compilation formellement certifiée. Ce chapitre expose les notions préliminaires nécessaires à la compréhension du reste du mémoire. Je présente tout d'abord quelques éléments de sémantiques formelles des langages de programmation et les liens avec la compilation certifiée (section 2.1). Le travail de certification formelle étant réalisé avec l'assistant de preuve Coq et des extraits des développements avec Coq étant présentés tout au long du mémoire, j'introduis également cet outil et ses langages (section 2.2). Ces deux sections reposent sur des exemples tirés du premier article qui s'est intéressé à la vérification d'une passe de compilation [37]. Dans cet article, l'objectif de McCarthy et Painter d'obtenir *in fine* des preuves vérifiées par ordinateur apparaît clairement par les détails qui sont apportés aux preuves, même si l'article en question ne présente pas de formalisation et de vérification à l'aide d'un outil logiciel.

Puis, je me place dans le contexte de la compilation de langages parallèles dont les sémantiques ont des modèles mémoires faibles. La section 2.3 est une introduction à ce type de modèle mémoire.

Enfin, j'introduis des notions spécifiques aux développements Coq et nécessaires à la compréhension de la suite tels que des définitions sur les systèmes de transitions en Coq dans la section 2.4.

2.1 Sémantique des langages de programmation et compilation vérifiée

2.1.1 Systèmes de transitions étiquetées

Il est possible d'avoir des sémantiques formelles des langages de programmation. On en distingue trois familles :

— les sémantiques dénotationnelles dans lesquelles on voit les programmes, procédures et fonctions comme des fonctions mathématiques sur des interprétations des entrées et des sorties ; on s'intéresse ainsi à ce que le code calcule, pas comment il le calcule ;

— les sémantiques axiomatiques dans lesquelles on établit la validité de propriétés du code,

— les sémantiques opérationnelles qui décrivent comment sont exécutés les programmes, le plus souvent sous forme de systèmes de transitions étiquetées entre les états successifs d'une machine abstraite exécutant le programme.

Plus formellement, un système de transitions étiquetées est un (Σ, E, R) où Σ est l'ensemble des états possibles, E un ensemble d'évènements ou étiquettes, et $R \subseteq \Sigma \times E \times \Sigma$ une relation de transition. On peut être amené à considérer I l'ensemble des états initiaux, et F l'ensemble des états finaux.

Dans l'article de McCarthy et Painter [37], le langage cible de la compilation est un tout petit langage machine :

> *The computer language into which these expressions are compiled is a single address computer with an accumulator, called ac, and four instructions: li (load immediate), load, sto (store) and add.*

La sémantique de ce langage peut être représentée par un système de transitions étiquetées. On considère un ensemble dénombrable d'adresses mémoire $\mathcal{A}$. Un état mémoire de la machine est une fonction totale $\sigma : \mathcal{A} \cup \{\texttt{ac}\} \to \mathbb{Z}$. $\sigma[a \leftarrow v]$ dénote l'état mémoire σ' défini par:

$$\begin{cases} \sigma'(a') &= \sigma(a) \quad \text{si } a' \neq a \\ \sigma'(a) &= v \end{cases}$$

L'ensemble des instructions est :

$$\{\texttt{li}\ v \mid v \in \mathbb{Z}\} \cup \{\texttt{load}\ a \mid a \in \mathcal{A}\} \cup \{\texttt{sto}\ a \mid a \in \mathcal{A}\} \cup \{\texttt{add}\ a \mid a \in \mathcal{A}\}$$

Un programme est une séquence d'instructions, où [] est la séquence vide, et si s est une séquence d'instructions et i une instruction $i :: s$ est une séquence d'instructions. On note $[x]$ pour $x :: []$ et $s_1 +\!\!+ s_2$ pour la concaténation de deux séquences. L'ensemble Σ est l'ensemble des couples (σ, s).

Si on s'intéresse aux échanges avec la mémoire (hors accumulateur), l'ensemble E des événements peut-être :

$$\{\tau\} \cup \{\textbf{read}\ a\ v \mid a \in \mathcal{A} \text{ et } v \in \mathbb{Z}\} \cup \{\textbf{write}\ a\ v \mid a \in \mathcal{A} \text{ et } v \in \mathbb{Z}\}$$

où τ dénote un évènement « silencieux », c'est-à-dire qu'on ne souhaite pas observer. Pour la relation de transition R, on note $R\ \sigma\ e\ \sigma'$ par $\sigma \xrightarrow{e} \sigma'$. Cette relation de transition est définie par :

$$(\sigma,\ (\texttt{li}\ v) :: s) \xrightarrow{\tau} (\sigma[\texttt{ac} \leftarrow v], s)$$

$$(\sigma,\ (\texttt{load}\ a) :: s) \xrightarrow{\textbf{read}\ a\ \sigma(a)} (\sigma[\texttt{ac} \leftarrow \sigma(a)], s)$$

$$(\sigma,\ (\texttt{sto}\ a) :: s) \xrightarrow{\textbf{write}\ a\ \sigma(\texttt{ac})} (\sigma[a \leftarrow \sigma(\texttt{ac})], s)$$

$$(\sigma,\ (\texttt{add}\ a) :: s) \xrightarrow{\textbf{write}\ a\ v} (\sigma[\texttt{ac} \leftarrow v], s)$$
$$\text{avec } v = \sigma(\texttt{ac}) + \sigma(a)$$

2.1.2 Compilation

Un compilateur est un logiciel qui transforme un programme écrit dans un langage dit source, en un programme dans un langage dit cible. Très souvent, pour des langages de programmation usuels, un compilateur est organisé en plu-

sieurs passes de compilation, faisant ainsi intervenir des langages intermédiaires jouant alternativement le rôle de langage cible et de langage source.

Dans l'article de McCarthy et Painter, le langage source est un langage d'expressions, défini par la grammaire : $e ::= c \mid x \mid e + e$ où $c \in \mathbb{Z}$ et x est une variable appartenant à un ensemble dénombrable de variables $\mathcal{V}$.

La passe de compilation peut être décrite formellement par la fonction inductive suivante, où A est un sous-ensemble fini de $\mathcal{A}$ et *map* est une fonction injective de $\mathcal{V}$ dans $\mathcal{A}$:

$$\begin{aligned} \mathcal{C}_A(c) &= [\texttt{li}\ c] \\ \mathcal{C}_A(x) &= [\texttt{load}\ map(x)] \\ \mathcal{C}_A(e_1 + e_2) &= \mathcal{C}_A(e_1) +\!\!+ [\texttt{sto}\ a] +\!\!+ \mathcal{C}_{A\cup\{a\}}(e_2) +\!\!+ [\texttt{add}\ a] \text{ où } a \notin A \end{aligned}$$

2.1.3 Compilation certifiée

L'essentiel du compilateur Compcert a été formellement certifié, c'est-à-dire qu'à chaque passe de compilation soit :

— la fonction de compilation a été écrite dans l'assistant de preuve Coq, et il a été prouvé avec Coq que la fonction de compilateur préserve la sémantique entre les programmes compilés et les programmes sources ;

— la fonction de compilation a été écrite avec un autre langage de programmation, mais en Coq a été écrit un validateur, qui étant donné un programme source et le programme cible associé, indique si la transformation effectuée extérieurement est valide, ainsi que la preuve que le validateur est correct.

Pour ce faire il est bien sûr indispensable de définir formellement ce que signifie préserver la sémantique lors de la compilation. Usuellement (mais pas toujours [6]) on considère les sémantiques opérationnelles du langage cible et du langage source, sous forme de systèmes de transitions étiquetées. L'exécution d'un programme est alors une séquence t, finie ou infinie, de transitions, telle que si (a, e_1, b) et (c, e_2, d) sont deux éléments successifs de t alors $R\ a\ e_1\ b$, $R\ c\ e_2\ d$ et $c = b$, et si le premier état du premier élément de t est un état initial. Si la trace est finie alors la dernière transition (a, e, b) de la séquence est telle que $\forall e, \forall c, \neg R\ b\ e\ c$. La trace d'une exécution est alors une séquence d'évènements résultant d'une exécution.

Comparer les sémantiques d'un programme source et d'un programme cible revient à comparer leurs traces. On peut exiger que les ensembles de traces soient strictement identiques, ou ne considérer que les évènements observables,

c'est-à-dire différents de τ, des traces. Ceci est toutefois une exigence trop forte pour un compilateur. En effet les sémantiques de référence des langages de programmation les plus utilisés sont sous spécifiées. Pour le langage C par exemple, elle ne précise pas l'ordre d'évaluation des expressions, mais autorise les compilateurs à faire un choix. On considère ainsi que la sémantique est préservée si l'ensemble des traces du programme cible est inclus dans l'ensemble des traces du programme source.

Enfin il n'est pas nécessaire de s'intéresser à la préservation sémantique de programmes erronés. On dira qu'un programme est erroné s'il a au moins un comportement (c'est-à-dire une trace) erroné. Une possibilité de définir un comportement erroné pouvant être de définir l'ensemble des états terminaux, et de caractériser un comportement erroné comme une exécution finie mais qui ne se termine pas dans un état terminal.

En notant $\mathit{safe}(p)$ le fait qu'un programme n'a pas de comportements erronés, et $\mathit{obs}_{\mathcal{L}}(p)$ l'ensemble des comportements observables de p pour la sémantique d'un langage $\mathcal{L}$, alors le résultat de préservation sémantique que l'on souhaite établir pour une étape de compilation $\mathcal{C}()$ entre un langage $\mathcal{L}_S$ et un langage $\mathcal{L}_T$ est :

$$\begin{aligned}\forall p \in \mathcal{L}_S, \quad & \mathit{safe}(p) \Rightarrow \\ & \forall o, o \in \mathit{obs}_{\mathcal{L}_T}(\mathcal{C}(p)) \Rightarrow o \in \mathit{obs}_{\mathcal{L}_S}(p).\end{aligned}$$

C'est ce qu'on appelle une simulation « en arrière ». Il est également possible pour les langages ayant une sémantique déterministe de prouver une simulation « en avant » c'est-à-dire que l'on teste l'inclusion de comportements observables du programme source dans les comportements du langage cible. Les simulations en avant sont plus faciles à prouver, mais ne peuvent être utilisées pour prouver la preuve de correction de passes de compilation de langages intrinsèquement non déterministes. Ces simulations ne conviennent donc pas aux langages que je traite dans ce travail.

Afin d'établir ces résultats sur les traces, il convient préalablement de prouver des résultats sur les exécutions des programmes. En particulier il est essentiel de définir des relations entre les états du langage source et du langage cible, pour pouvoir exprimer des énoncés de la forme :

> Soit a' et b' des états de $\mathcal{L}_T$, un évènement e, et un état a de $\mathcal{L}_S$ tels que $R_T\ a'\ e\ b'$ et $a \equiv a'$ alors il existe b un état de $\mathcal{L}_S$ tel que $R_S\ a\ e\ b$.

Cet énoncé est un résultat de simulation à un pas. C'est une hypothèse forte, rarement vérifiée dans les passes de compilation, et il est en pratique nécessaire de considérer des hypothèses plus faibles. Par exemple la simulation option est

$$\frac{A \in \Gamma}{\Gamma \vdash A} \tag{2.1}$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \tag{2.2}$$

$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \to B} \tag{2.3}$$

FIGURE 2.1 – *Déduction naturelle*

définie par : pour tout programme p sûr, tout état a' et b' de $\mathcal{C}(p)$, évènement e tels que $R_T\ a'\ e\ b'$, et tout état a de p équivalent à a', soit on a la simulation à un pas, soit pour une mesure $|\cdot|$ et une relation $<$ bien fondée on a $|a'| < |b'|$ et a est équivalent à b'. La seconde alternative de la simulation option permet d'éviter que le programme compilé entre dans une boucle infinie d'évènements silencieux.

2.2 L'assistant de preuve Coq

2.2.1 Présentation générale

L'assistant de preuve Coq [51] est un outil logiciel permettant l'écriture assistée et la vérification de preuves. Il est basé sur la correspondance de Curry-Howard [26] qui relie des arbres de preuve en déduction naturelle et des termes de λ-calculs typé. La figure 2.1 présente le système d'inférence pour un calcul propositionnel avec l'implication comme seul connecteur. Les propositions atomiques et les formules sont notées en majuscules et Γ est un ensemble de propositions. Le jugement $\Gamma \vdash F$ où F est une formule obtenue par la grammaire $F ::= A \mid F \to F$, indique la validité de la formule F sous les hypothèses Γ.

La figure 2.2 présente le système d'inférence pour le typage d'un λ-calcul. Les types primitifs et les types sont notés en majuscule, et les termes du calcul sont définis par la grammaire : $t ::= x \mid t\ t \mid \lambda(x{:}T).t$ où x dénote une variable et T est un type dont la grammaire est la même que les formules de la déduction naturelle. Nous identifions les termes à un renommage des variables liées près. Le jugement $\Gamma \vdash t : T$ indique que sous l'environnement de typage Γ, qui associe un type donné à une variable donnée, le terme t a le type T.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \tag{2.4}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B} \tag{2.5}$$

$$\frac{\Gamma \cup \{x : A\} \vdash t : B}{\Gamma \vdash \lambda(x{:}A).t : A \to B} \tag{2.6}$$

FIGURE 2.2 – *λ-calcul simplement typé*

Intuitivement, comme il y a correspondance entre les règles de ces deux systèmes d'inférence, on comprend qu'à un arbre de preuve de la déduction naturelle, peut correspondre un arbre de typage d'un λ-terme. L'arbre de typage ne dépend que du λ-terme. On peut ainsi faire correspondre : à un énoncé de la logique un *type* dans le λ-calcul, et à une preuve d'un énoncé un *terme* du λ-calcul.

Le calcul des constructions [20] est un λ-calcul typé avec un système de types très riche. En plus des fonctions usuelles et de leurs types, des propriétés logiques d'ordre supérieur peuvent être exprimées dans ce λ-calcul. Il a été au cours du temps enrichi pour permettre des définitions inductives [21] et coinductives [23]. Le calcul *Predicative Calculus of (co)Inductive Constructions* qui est issu de ces recherches est à la base de l'assistant de preuve Coq.

Dans cette section nous donnons les éléments qui permettent de comprendre les extraits des développements Coq que nous avons réalisés. Pour des expositions plus complètes de Coq, nous renvoyons le lecteur aux ouvrages de Bertot et Castéran [10], de Chlipala [19] ou au cours en ligne de Pierce et al. [17]. Une autre présentation concise de Coq est celle de Bertot [9].

Nous présentons les aspects de l'assistant de preuve Coq dont nous avons besoin dans la suite de la présentation à travers la formalisation de l'article de MacCarthy et Painter [37]. Le langage d'entrée considéré est un simple langage d'expressions arithmétiques avec des variables, des constantes et une seule opération : l'addition. Le langage cible est un petit langage machine avec des instructions de chargement (depuis la mémoire/les registres ou une constante), d'affectation et d'addition avec un registre d'accumulation. L'état pour l'évaluation d'expressions du langage d'entrée est un environnement qui associe des valeurs aux variables, tandis que pour le langage cible il s'agit d'une fonction totale des adresses mémoire vers des valeurs. L'équivalence d'états est faite par une

association entre variables et adresses mémoires : deux états source et cible sont équivalents si les valeurs associées aux variables/adresses liées sont les mêmes. Une fonction d'évaluation est donnée pour chacun des langages et le théorème principal de l'article énonce que partant de deux états équivalents l'évaluation d'une expression ou l'évaluation du code issu de la compilation donne le même résultat.

Nous avons formalisé cet article en Coq. La seule hypothèse que nous avons dû ajouter est que l'accumulateur ne doit pas être lié avec une variable par l'équivalence d'états.

2.2.2 Définitions et définitions inductives

Un développement Coq est constitué d'un ensemble de *bibliothèques*. Une bibliothèque est simplement l'ensemble des éléments présents dans un fichier. Ainsi le fichier `List.v` (puis sa forme compilée `List.vo`) constitue la bibliothèque List. Pour pouvoir utiliser les éléments d'une bibliothèque déjà compilée, il est nécessaire de faire usage de la commande Require suivie d'une liste non vide de noms de bibliothèques. Un élément e d'une bibliothèque Lib ainsi requise peut alors être utilisé avec le nom Lib.e. Pour pouvoir omettre ce préfixe, on peut importer la bibliothèque avec la commande Import. Enfin ces deux commandes peuvent être combinées pour requérir et importer une liste de bibliothèques.

La formalisation de l'article de McCarthy et Painter commence ainsi par :

```
Require Import List Arith ZArith.
```

Les bibliothèques List, Arith et ZArith font partie des bibliothèques standards de Coq et contiennent respectivement des formalisations d'une structure de données polymorphe de liste, d'arithmétique sur les entiers naturels (Peano) et d'arithmétique sur les entiers relatifs.

Nous importons ensuite ListNotations. Ceci n'est pas une bibliothèque mais un *module* qui est défini dans la bibliothèque List. Un module est également un ensemble d'éléments mais est nommé explicitement et défini à l'intérieur d'une bibliothèque. Comme nous le verrons plus tard, un module peut être paramétré par d'autres modules. Il n'y a pas besoin de requérir un module, mais pour éviter de préfixer les éléments du module, on peut importer un module comme on peut importer une bibliothèque.

```
Import ListNotations.
```

Les commentaires en Coq sont formés comme en OCaml à l'aide des délimiteurs *(* *)*. Les commentaires commençant par *(*** sont utilisés par l'outil `coqdoc` pour la génération de documentations dans divers formats :

```
(** * Formalisation in Coq of "CORRECTNESS OF A COMPILER FOR
    ARITHMETIC EXPRESSIONS" by McCarthy and Painter, 1967 *)
(** ** The source language *)
```

Nous commençons par formaliser le langage source. Comme nous souhaitons utiliser des noms communs au langage source et au langage cible, nous choisissons de définir le langage source dans un module nommé Source que nous n'importerons pas. Les modules sont utilisés ici comme espaces de nommage :

```
Module Source.
```

Toute définition en Coq comprend un nom, un type et un terme. Ainsi nous modélisons une variable du langage source comme un entier naturel :

```
Definition variable : Set := nat.
```

En Coq toutes les définitions définissent des termes, et tout terme a un type qui est lui-même un terme. Le type prédéfini nat est un type sur lequel on peut faire des calculs : son type, encore appelé sorte, est Set. Le type de Set est Type, et le type de Type est Type. En réalité, en interne, Coq utilise une hiérarchie infinie de sortes Type_i et une valeur de type Type_i ne peut utiliser pour être construite que des valeurs de type Type_j avec $j < i$. Jusqu'à récemment une valeur spécifique de i était fixée au moment de la définition. Les versions récentes de Coq introduisent la notion de polymorphisme d'univers [50].

Les types comme nat sont définis de manière inductive. Nous définissons ici un nouveau type t qui fait également partie du monde calculatoire. Les valeurs de type t peuvent être construites à l'aide des *constructeurs* du type, var, const et add. Par exemple le constructeur var prend en argument une variable et retourne une valeur de type t. Les expressions de notre langage source sont donc des variables, des constantes ici modélisées par des entiers relatifs, et enfin l'application de l'opération d'addition à deux expressions du langage :

```
Inductive t : Set :=
| var   : variable → t
| const : Z → t
| add   : t → t → t.
```

Il est également possible de définir des éléments qui n'appartiennent pas au monde calculatoire mais au monde logique. Pour cela nous indiquons que le type

est Prop (dont le type est également Type). Dans la définition suivante, l'inductif In a le type variable→ t→Prop. Il prend en argument une variable et une expression et renvoie une valeur logique : c'est en fait un prédicat défini inductivement :

```
Inductive In : variable → t → Prop :=
| in_var : ∀ (v:variable), In v (var v)
| in_add : ∀ (v:variable) (e1 e2:t), In v e1 ∨  In v e2 → In v (add e1 e2).
```

Si les constructeurs du type inductif t sont semblables à des fonctions usuelles de langages fonctionnels de programmation, le type des constructeurs de In sont différents. En effet, le premier constructeur sert à formaliser le fait qu'une variable v appartient bien à l'expression var v. Ce fait s'écrit In v (var v). Or ce type dépend non pas du *type* du premier argument du constructeur, mais de la *valeur* de cet argument. C'est ce qu'on appelle un *produit dépendant* et qui est noté ∀ (x:A),B où B est un terme qui peut contenir x en variable libre. Si jamais B ne dépend pas de x, on écrit A→B qui correspond au type classique d'une fonction.

Dans les définitions, il est possible d'omettre des informations de types qui sont alors inférées dans Coq. Ainsi on peut aussi bien écrire :

```
Inductive In : variable → t → Prop :=
| in_var : ∀ v, In v (var v)
| in_add : ∀ v e1 e2, In v e1 ∨  In v e2 → In v (add e1 e2).
```

La définition suivante omet également le type du terme défini. Un état est une fonction totale des variables dans les valeurs du langage source (ici modélisées par des entiers relatifs) :

```
Definition state := variable → Z.
```

Les définitions suivantes de la fonction c sont équivalentes :

```
Definition c : nat → state → Z := fun x s ⇒ s x.
Definition c := fun (x:nat) (s:state) ⇒ s x.
Definition c (x:nat) (s:state) : Z := s x.
```

Enfin nous définissons par induction l'évaluation d'une expression dans un environnement :

```
Fixpoint value (e:t) (env:state) : Z :=
  match e with
    | var x ⇒ c x env
    | const v ⇒ v
    | add e1 e2 ⇒ ( (value e1 env) + (value e2 env) )%Z
  end.
```

Les fonctions inductives en Coq doivent forcément terminer. Un critère simple pour l'assurer est que les appels récursifs des fonctions se fassent sur des sous-termes syntaxiques stricts de l'un des arguments. L'utilisateur peut préciser quel argument est concerné ou laisser Coq le déterminer. Cette définition fait également appel à du filtrage : pour une valeur d'un type inductif, le seul moyen de la construire est d'utiliser l'un des constructeurs du type inductif. Le filtrage permet de raisonner par cas selon le constructeur utilisé, et permet de décomposer la valeur en récupérant les arguments du constructeur utilisé. En Coq un filtrage doit être complet, c'est-à-dire considérer tous les constructeurs possibles du type inductif.

Coq permet de définir des notations, en particulier infixes, et des portées pour ces notations. Ainsi la bibliothèque ZArith contient la définition de la fonction Z.add d'addition de deux entiers relatifs, la notation infixe + pour cette fonction, et la portée de notation %Z.

La fin de la définition d'un module est indiquée avec End :

```
End Source.
```

Le début de la définition du module concernant la modélisation du langage cible n'utilise que des notions de Coq déjà rencontrées pour le langage source :

```
(** ** The target language *)
Module Target.

 Definition address : Set := option nat.

 Definition ac : address := None. (* The accumulator. *)

 Inductive lt : address → address → Prop :=
 | lt_ac : ∀ l, lt ac (Some l)
 | lt_some : ∀ n1 n2, Peano.lt n1 n2 → lt (Some n1) (Some n2).

 Inductive instruction : Set :=
 | li : Z → instruction
 | load : address → instruction
 | store : address → instruction
 | add : address → instruction.

 Definition t := list instruction.

 Definition state := address → Z.
```

```
Definition c (x:address) (s:state) := s x.
```

Trois éléments sont à détailler :

— option est un type pré-défini dans Coq par :

```
Inductive option (A : Type) : Type :=
 | Some : A → option A
 | None : option A.
```

— list est un type pré-défini par :

```
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A → list A → list A.
```

— Peano.lt est la relation d'ordre $<$ sur les entiers naturels nat.

2.2.3 Théorèmes et preuves

L'élément suivant du module Target est nommé lt_trans et son type est $\forall\ a_1\ a_2\ a_3$, lt $a_1\ a_2 \rightarrow$ lt $a_2\ a_3 \rightarrow$ lt $a_1\ a_3$. Le terme toutefois n'est pas donné, et cette définition est introduite par le mot-clé Lemma. Lorsque ce lemme est donné à l'assistant de preuve Coq, il entre dans un mode interactif de preuve. Il s'attend à recevoir des instructions appelées tactiques pour le guider dans la construction d'un terme ayant le type donné, c'est-à-dire une preuve de ce type vu comme un énoncé.

Je commente ici le script de preuve de ce lemme pour donner au lecteur néophyte un aperçu de l'écriture d'un tel script, mais l'objectif n'est pas une présentation exhaustive de l'ensemble des tactiques utilisés dans mes développement Coq.

Le mot-clé Proof est optionnel est n'est utilisé que pour des raisons esthétiques :

```
Lemma lt_trans :
 ∀ a1 a2 a3, lt a1 a2 → lt a2 a3 → lt a1 a3.
Proof.
```

Le mode de preuve interactive indique l'ensemble des buts à prouver et les éléments présents dans le contexte. Ainsi Coq affiche :

```
1 subgoals, subgoal 1 (ID 155)

  ============================
   forall a1 a2 a3 : address, lt a1 a2 -> lt a2 a3 -> lt a1 a3
```

Ce qui est au dessus de la barre peut être considéré comme le Γ dans le système d'inférence de la déduction naturelle. Les tactiques les plus simples sont essentiellement des applications de règles de la déduction naturelle pour la logique d'ordre supérieure associée au calcul des constructions inductives. Ainsi la tactique intro applique la règle d'introduction de la quantification universelle ou d'introduction de l'implication. Cette tactique prend en argument optionnel un nom de terme. La tactique dérivée intros introduit tant que c'est possible lorsqu'elle n'a pas d'arguments, ou autant de fois qu'il y a de noms données en paramètre. Ainsi après la tactique,

```
intros a1 a2 a3 H1 H2.
```

l'état courant de la preuve devient :

```
1 subgoals, subgoal 1 (ID 160)

  a1 : address
  a2 : address
  a3 : address
  H1 : lt a1 a2
  H2 : lt a2 a3
  ============================
   lt a1 a3
```

Il faut alors raisonner par cas sur l'adresse a1 (qui peut être une adresse qui n'est pas l'accumulateur ou None c'est-à-dire l'accumulateur). Dans le premier cas, l'introduction d'une nouvelle variable est nécessaire : nous l'appelons a1. Après la tactique

```
case a1 as [a1 | ].
```

le seul sous-but est remplacé par deux sous-buts, le premier dans lequel a1 a été remplacé par Some a1 et le second dans lequel il a été remplacé par None :

```
2 subgoals, subgoal 1 (ID 178)

  a1 : nat
  a2 : address
  a3 : address
  H1 : lt (Some a1) a2
  H2 : lt a2 a3
  ============================
   lt (Some a1) a3

subgoal 2 (ID 180) is:
 lt None a3
```

Afin de structurer la preuve, il est pratique d'utiliser la fonctionnalité de « *bullet* » de Coq. Dès que l'on a plus d'un sous-but et que l'on utilise un de des symboles $-$, + ou $*$, le même symbole doit être utilisé pour tous les autres sous-buts existants au moment de la première utilisation.

Pour ce qui est de la tactique qui suit le symbole $-$, inversion est à utiliser. Cette tactique prend en argument un identifiant (présent dans le contexte) dont le type est un type inductif, et pour chaque constructeur du type inductif ajoute toutes les conditions nécessaires qui doivent être vérifiées pour que l'hypothèse puisse être prouvée par le constructeur. Les cas qui sont immédiatement visiblement impossibles sont éliminés : c'est le cas d'égalités entre constructeurs différents. Cette tactique introduisant des nouvelles hypothèses dans le contexte, on peut comme dans le cas de case explicitement les nommer. Ainsi après

```
− inversion H1 as [ | a1' a2' H1'].
```

on obtient l'état courant suivant :

```
a1 : nat
 a2 : address
 a3 : address
 H1 : lt (Some a1) a2
 H2 : lt a2 a3
 a1' : nat
 a2' : nat
 H1' : a1 < a2'
 H0 : a1' = a1
 H3 : Some a2' = a2
 ============================
  lt (Some a1) a3
```

En effet, bien que lt ait deux constructeurs, l'égalité nommée H0 ici, serait pour le cas du constructeur lt_ac, None = Some a1, ce qui est impossible. Ce cas est donc éliminé automatiquement. La tactique inversion génère plusieurs égalités qui sont des simples renommages et d'autres égalités entre une variable et un terme qui n'est pas une simple variable. La tactiques subst permet de réécrire les secondes partout et d'éliminer les renommages. On obtient alors :

```
1 focused subgoals (unfocused: 1)
, subgoal 1 (ID 229)

  a1 : nat
  a3 : address
  a2' : nat
```

```
  H1' : a1 < a2'
  H1 : lt (Some a1) (Some a2')
  H2 : lt (Some a2') a3
  ============================
   lt (Some a1) a3
```

On procède de même avec l'hypothèse H2 :

```
inversion H2 as [ | a1' a3' H2']; subst.
```

Notons qu'ici nous avons utiliser le symbole ; plutôt que . entre les deux tactiques. Dans ce cas la seconde tactique est appliquée à tous les sous-buts générés par la première (ici un seul) et la composition des deux tactiques est exécutée en bloc. On obtient :

```
1 focused subgoals (unfocused: 1)
, subgoal 1 (ID 280)

  a1 : nat
  a2' : nat
  H1' : a1 < a2'
  H1 : lt (Some a1) (Some a2')
  a3' : nat
  H2' : a2' < a3'
  H2 : lt (Some a2') (Some a3')
  ============================
   lt (Some a1) (Some a3')
```

On peut alors appliquer l'un des constructeurs de lt. Plutôt que de le faire explicitement, on fait appel à la tactique constructor qui applique le premier constructeur possible :

```
constructor.
```

Il reste alors à prouver :

```
  a1 : nat
  a2' : nat
  H1' : a1 < a2'
  H1 : lt (Some a1) (Some a2')
  a3' : nat
  H2' : a2' < a3'
  H2 : lt (Some a2') (Some a3')
  ============================
   a1 < a3'
```

On peut facilement conclure par transitivité de < :

```
transitivity a2'; assumption.
```

la tactique assumption indiquant simplement que le but est exactement une des hypothèses. Ce sous-but est prouvé :

```
1 subgoals, subgoal 1 (ID 180)

subgoal 1 (ID 180) is:
 lt None a3
```

Il faut passer au sous-but restant qui est prouvé de manière très similaire :

```
  – inversion H1; inversion H2; subst;
    constructor.
```

La fin du script de preuve, obligatoire, est indiquée par la commande Qed. C'est à ce moment là que le terme de preuve construit avec les tactiques est vérifié : le noyau de Coq vérifie que l'énoncé du théorème est bien le type du terme de preuve construit. Dans le cas où une tactique produirait des termes erronés, un terme de preuve incorrect ne serait donc pas ajouté à l'environnement global de Coq.

```
Qed.
```

La logique de Coq ne contient pas le tiers-exclu. Dès lors on ne peut pas avoir un raisonnement de la forme : soit deux adresses mémoires sont égales, soit elles sont différentes, tant que l'on ne montre pas que l'égalité entre adresses est décidable. Une possibilité pour exprimer une telle propriété est :

```
Lemma eq_decidable (l l' : address) : l = l' ∨ ~ l = l'.
```

ou $\vee$ est une notation pour :

```
Inductive or (A B : Prop) : Prop :=
| or_introl : A → or A B
| or_intror : B → or A B.
```

Le problème avec cette version est que eq_decidable l l' est alors de type Prop. Comme il n'est possible pas en Coq que le résultat d'un calcul dépende d'une preuve (c'est une forme faible d'indiscernabilité des preuves), on ne peut pas utiliser ce lemme et le filtrage de motif pour écrire une fonction dont le flot de contrôle dépend de l'égalité de deux adresses.

Une première solution pour résoudre ce problème est de définir une fonction Booléenne qui teste l'égalité de deux adresses (et de montrer son équivalence avec la relation d'égalité). Une seconde solution est d'utiliser le type sumbool défini par :

```
Inductive sumbool (A B : Prop) : Set :=
| left  : A → sumbool A B
| right : B → sumbool A B.
```

sumbool A B, noté {A}+{B}, a pour type Set. Il est donc possible du faire du filtrage de motif sur des valeurs de ce type. En Coq, dans la construction conditionnelle if then else, la condition peut être n'importe quelle valeur dont le type inductif dans Set a deux constructeurs. On peut donc utiliser dans une telle construction le lemme de décidabilité de l'égalité sur les adresses exprimé à l'aide de sumbool :

```
Lemma eq_dec (l l' : address) : { l = l' } + { ~ l = l' }.
Proof. repeat (decide equality). Qed.
```

ce qui permet de définir la fonction a qui met à jour un état mémoire avec une nouvelle valeur pour l'adresse l :

```
Definition a (l:address)(v:Z)(s:state) : state :=
 fun l' ⇒ if eq_dec l' l then v else s l'.
```

J'omets ici les quelques lemmes sur la fonction a, le code complet est présent à l'annexe A.

Plutôt que définir la sémantique de ce langage sous forme d'une relation de transition étiquetée, je la définis sous forme d'une fonction entre états mémoires :

```
Fixpoint step (i:instruction)(s:state) : state :=
 match i with
  | li v     ⇒ a ac v s
  | load adr  ⇒ a ac  (c adr s) s
  | store adr ⇒ a adr (c ac s)  s
  | add adr   ⇒ a ac  ((c adr s) + (c ac s))%Z s
 end.
```

Pour un programme complet, il suffit d'itérer la fonction précédente :

```
Fixpoint outcome (p : t) (s:state) : state :=
 match p with
  | [] ⇒ s
  | first::rest ⇒ outcome rest (step first s)
 end.
```

Enfin, en suivant l'article de McCarthy et Painter, il est nécessaire de définir une équivalence entre états, mais seulement sur un sous-ensemble d'adresses qui est représenté ici par une relation P unaire :

```
Definition eqt (P:address→Prop)(s1 s2 : state) : Prop :=
 ∀ l, P l → c l s1 = c l s2.
```

```
End Target.
```

Tout est alors prêt pour définir la fonction de compilation et énoncer le théorème de correction de cette compilation. Nous définissons ceci dans une section Coq. Ceci permet de poser des hypothèses à l'aide des mot-clés Variable et Hypothesis. À la fermeture de la section, une quantification universelle est ajoutée pour chacun de ces éléments à toutes les définitions qui les suivent dans la section.

Nous introduisons ici une fonction de traduction loc entre les variables et les adresses, (appelée *map* dans la section précédente), et le fait ac n'est pas dans l'image de cette fonction :

```
(** ** Compilation *)
Section compilation.

Variable loc : Source.variable → Target.address.
Hypothesis loc_prop : ∀ x, loc x <> Target.ac.
```

La définition de la fonction compile est alors extrêmement proche de la version donnée dans la section précédent, si ce n'est que l'ensemble des adresses utilisées A est remplacé par la valeur de l'adresse la plus grande t :

```
Fixpoint compile (e : Source.t) (t : nat) : Target.t :=
  match e with
    | Source.const v ⇒ [ Target.li v ]
    | Source.var x   ⇒ [ Target.load (loc x) ]
    | Source.add e1 e2 ⇒
      (compile e1 t) ++ [Target.store (Some t)] ++
      (compile e2 (t+1)) ++ [Target.add (Some t)]
  end.
```

Une adresse valide est alors une adresse qui est plus petite qu'une adresse donnée :

```
Definition valid (t:Target.address) := fun l ⇒ Target.lt l t.
```

et le théorème de correction s'énonce comme suit :

```
Theorem compiler_correctness:
  ∀ (e:Source.t)(env:Source.state)(s:Target.state)(t:nat),
    (∀ x, Source.In x e → Target.lt (loc x) (Some t)) →
    (∀ x, Source.In x e → Source.c x env = Target.c (loc x) s) →
    Target.eqt (valid (Some t))
               (Target.outcome (compile e t) s)
```

```
            (Target.a Target.ac (Source.value e env) s).
 Proof. (* omit *) Qed.
End compilation.
```

Notons l'utilisation de la commande Theorem qui est un synonyme de Lemma, tout comme Proposition et Fact.

2.2.4 Modules

Dans la section précédente nous avons utilisé les modules en tant qu'espaces de nommage. Le système de module en Coq est beaucoup plus expressif et est un support à l'abstraction.

Tout d'abord, il est possible de définir des types de modules. Par exemple pour définir un type de module qui contient un type sur lequel l'égalité est décidable, on peut écrire:

```
Module Type DecidableType.
 Parameter t : Type.
 Axiom eq_dec : ∀(x y :t), {x=y}+{~x=y}.
End DecidableType.
```

Lorsqu'un type de module contient des paramètres ou des axiomes, c'est-à-dire seulement un nom et un type mais pas de terme associé, alors les modules réalisant cette signature doivent fournir un terme pour chacun des paramètres et axiomes.

Un module de type peut contenir également des définitions complètes : dans ce cas les modules l'implantant doivent fournir exactement les mêmes termes que dans le type de module.

Lors de la définition d'un module, on peut indiquer que celui-ci doit implanter la signature d'un type de module, par exemple :

```
Module Natural : DecidableType.
 Definition t := nat.
 Definition eq_dec := eq_nat_dec.
End Natural.
```

Dans ce cas, tous les éléments en plus de t et eq_dec qui pourraient être contenus dans Natural ne sont pas visibles de l'extérieur. En utilisant <: plutôt que :, le système vérifie simplement que le module implante la signature, mais le module peut fournir des fonctionnalités supplémentaires.

Il est possible d'avoir des modules (et des types de modules) d'ordre supérieur. Par exemple, on peut vouloir ajouter le fait qu'un type décidable a une égalité booléenne qui est équivalent à l'égalité :

```
Module Type Eqb (Import T : DecidableType).
 Parameter eqb : t → t → bool.
 Parameter eqb_eq : ∀ x y, eqb x y = true <→ x = y.
End Eqb.
```

Lorsque l'on veut s'abstraire de détails d'implantation, on peut donc placer un développement à l'intérieur d'un module qui prend en argument des modules pour les aspects que l'on veut abstraire. Il faut bien sûr fournir des types de modules pour ces abstractions.

Si on se contente d'abstraire ainsi, cela revient au même qu'ajouter des paramètres et axiomes à un développement Coq. Il convient alors de fournir pour chacun des types de modules, un module sans axiomes implantant ce type. On est alors sûr que les axiomes contenus dans les types de modules ne rendent pas la logique de Coq incohérente.

Dans la suite de ce mémoire, je fais un grand usage du système de modules de Coq.

2.3 Modèles mémoire faibles

De nombreuses optimisations, en particulier des processeurs, ne sont pas observables par les utilisateurs, à partir du moment où le programme considéré est séquentiel. Avec l'introduction du parallélisme programmable, il y a désormais plusieurs fils d'exécution sur plusieurs cœurs au sein d'un seul processeur. Ces optimisations deviennent alors observables par les utilisateurs, et rendent la sémantique des processeurs complexe, en particulier ce qui concerne le comportement de la mémoire. Les processeurs ne sont pas les seuls à réaliser de telles optimisation, les compilateurs peuvent également modifier la structure du programme à des fins d'optimisation. Dans tous les cas, le modèle mémoire peut être vu comme un contrat entre le concepteur du processeur ou du compilateur et le programmeur. Il décrit le fonctionnement des opérations mémoires et propose un ensemble de conditions à respecter afin de pouvoir en ignorer les aspects les plus complexes.

Un modèle mémoire intuitif en contexte parallèle est celui de la *consistance séquentielle*. Informellement les comportements de deux programmes mis en parallèle sont les entrelacements des comportements des deux programmes considérés séparément. Par exemple si on considère que les deux variables en mémoire `x` et `y` contiennent initialement la valeur `0` et que les $\mathtt{r}_i$ sont des registres du processeur, le programme suivant :

Cœur 1	Cœur 2
$\mathtt{x} = 1;$	$\mathtt{y} = 1;$
$\mathtt{r}_0 = \mathtt{y};$	$\mathtt{r}_1 = \mathtt{x};$

ne peut donner avec un modèle mémoire séquentiellement consistant, uniquement :

— $\mathtt{r}_0 = 0$ et $\mathtt{r}_1 = 1$,
— ou $\mathtt{r}_0 = 1$ et $\mathtt{r}_1 = 0$,
— ou $\mathtt{r}_0 = 1$ et $\mathtt{r}_1 = 1$

qui résultent de tous les entrelacements possibles des instructions.

Or sur la plupart des processeurs actuels, on peut également observer le résultat $\mathtt{r}_0 = 0$ et $\mathtt{r}_1 = 0$. En effet si on considère le programme de chaque cœur indépendamment, il y a pas de dépendance de données entre la première et la seconde instruction. De ce point de vue, l'optimisation classique qui consiste à inverser les deux instructions, est tout à fait valide, et ne peut pas être observée dans un programme séquentiel. Caractériser de tels comportements de la mémoire revient à proposer des modèles moins contraints que la consistance séquentielle, que l'on appelle *modèles mémoire faibles* [2]. Ces dernières années de nombreux travaux se sont intéressés à la formalisation des modèles mémoires des processeurs les plus répandus [3, 46, 41, 49] mais également des modèles mémoires pour les langages de programmation répandus avec concurrence [35, 14].

Une propriété fondamentale des modèles mémoire faibles existants, est que si aucune exécution séquentiellement consistante d'un programme parallèle ne contient de *data race*, c'est-à-dire deux accès concurrents à la même adresse mémoire dont l'un au moins est une écriture, alors toutes les exécutions de ce programme sont séquentiellement consistantes [45].

Or si cette condition est suffisante, elle n'est pas nécessaire, et il existe des conditions plus faibles que la *data race freeness* (DRF) qui permettent néanmoins d'assurer le comportement séquentiellement consistant de programmes [40]. En pratique, notamment pour les applications dont l'efficacité est critique, comme les noyaux des systèmes d'exploitation, il est classique de considérer des pro-

grammes qui n'ont pas la DRF. La vérification de ce type de programmes étant d'un grand intérêt, des travaux très récents s'intéressent à la vérification de programmes en tenant compte de modèles mémoires faibles [55].

De même il est important de prendre en compte les modèles mémoires faibles pour la compilation vérifiée, d'une part pour être sûr que les optimisations classiques restent corrects dans le modèle mémoire faible considéré, et d'autre part pour introduire des optimisations spécifiques aux processeurs multi-cœurs tel que l'optimisation des synchronisations.

2.4 Sémantique en Coq

Dans la suite, on présente les notions préliminaires relatives aux définitions en Coq des propriétés que l'on utilise dans les chapitres qui suivent. Une partie de ces propriétés et définitions existent déjà dans les développements sur lesquels on se base (CompcertTSO) et on va donc différencier les choses déjà existantes en Coq (sur fond gris foncé) des choses que l'on a adapté ou créé en Coq (sur fond gris clair).

Systèmes de transitions Dans la suite, je vais utiliser des prédicats sur les états et les systèmes de transition pour simplifier l'énoncé des propriétés de simulation en Coq et dans ce mémoire.

Définition 1 (taustar) *taustar permet de considérer une suite finie d'événements silencieux.*

Dans le code Coq, on utilisera des prédicats différents dans chacune des différentes machines. On a d'abord tenté d'utiliser une définition paramétrée par un type d'événement donné. Mais le système d'inférence de Coq posait des problèmes lors de l'utilisation de certaines techniques automatiques (eauto, eapply) car le système d'inférence ne reconnaissait pas les bons paramètres et on était obligé de le fournir explicitement à chaque appel de lemmes (ce qui était laborieux).

— Pour le modèle mémoire, l'événement silencieux est MMtau. Le prédicat permet de représenter les suites finis d'événement silencieux :

```
Inductive mstar {state: Type} (mstep: state → MMevent → state → Prop):
 state → state → Prop  :=
 | mstar_refl: ∀ ttso, mstar mstep ttso ttso
```

```
| mstar_cons: ∀ ttso ttso' ttso'', mstar mstep ttso' ttso'' →
  mstep ttso MMtau ttso' → mstar mstep ttso ttso''.
```

— pour l'ordonnanceur, on en distingue plusieurs car on a choisi d'utiliser différents événements silencieux pour représenter différentes étapes de ces systèmes de transitions étiquetés. On en parle plus en détail à la section 4.5. On distingue SCstep qui autorise l'exécution d'un processus léger et SCtau qui est un événement silencieux de la machine. On distingue donc les suites finies d'événements silencieux et les suites d'événements silencieux ou événement silencieux de l'ordonnanceur :

```
Inductive schedstar :=
 | schedstar_refl: ∀ s, schedstar scheduler_step s s
 | schedstar_cons: ∀ s s' s'', schedstar scheduler_step s' s''→
   scheduler_step s SCtau s' → schedstar scheduler_step s s''.
```

```
Definition tau_or_step ev : Prop :=
match ev with
| SCtau ⇒ True
| SCstep tid ⇒ True
| _ ⇒ False
end.

Inductive schedstarstep :=
 | schedstarstep_refl : ∀ s, schedstarstep scheduler_step s s
 | schedstarstep_cons : ∀ s s' s'' ev, tau_or_step ev →
                schedstarstep scheduler_step s' s'' →
                scheduler_step s ev s' →
                schedstarstep scheduler_step s s''.
```

— pour la machine globale et la machine par processus légers, on distingue également plusieurs événements silencieux différents ce qui donne une définition moins naturelle que ce que l'on voudrait :

```
Inductive tauscstar :=
   tauscstar_refl : ∀ s, tauscstar s s
 | tauscstar_sched :
    ∀ s s' s'' l, tau_sys _ l → stepr _ s l s' → tauscstar s' s'' → tauscstar s s''
 | tauscstar_tso :
    ∀ s s' s'' l, tau_tso _ l → stepr ts s l s' → tauscstar s' s'' → tauscstar s s''
 | tauscstar_tau :
    ∀ l s s' s'', tau _ l → stepr ts s l s' → tauscstar s' s'' → tauscstar s s''.
```

— pour la machine globale (mais uniquement des événements silencieux de l'ordonnanceur) :

```
Inductive tauschedstar : St ts → St ts → Prop :=
 | tauschedstar_refl: ∀ s, tauschedstar s s
 | tauschedstar_step:
  ∀ {s s' s'' l}, tau_sys _ l →
    stepr _ s l s' →
    tauschedstar s' s'' →
    tauschedstar s s''.
```

Définition 2 (tsotstep) *taustep représente un nombre fini d'événements silencieux suivis d'un événement donné ev.*

On distingue là aussi plusieurs définitions pour les différentes machines.

— Pour la machine mémoire :

```
Definition tsotstep {tso: Type} tso_step (ttso: tso) ev ttso'' :=
 ∃ ttso', ttsostar tso_step ttso ttso' ∧  tso_step ttso' ev ttso''.
```

— pour la machine globale, on interdit que l'événement soit un événement silencieux de l'ordonnanceur pour ne considérer que les cas qui représentent une avancée de l'exécution du programme,

```
Definition taustep s l s' : Prop :=
 ∃ s1, tauscstar s s1 ∧  stepr _ s1 l s' ∧  ∼ tau_sys _ l.
```

— pour la machine par processus légers :

```
Definition taustep_t s l s' :=
 ∃ s1, taustar ts s s1 ∧  stepr ts s1 l s'.
```

Définition 3 (pseudotau) *pseudotau est un prédicat représentant un événement de lecture en mémoire dans le langage de la machine par processus légers. On peut en théorie représenter d'autres instructions avec ce prédicat et son instance (en conservant potentiellement les preuves) dans la machine par processus légers est tec_pseudotau.*

```
match e with
  | THmemc (MEread _ _ _) ⇒ True
  | _ ⇒ False
end.
```

Définition 4 (fencetau) *fencetau est une prédicat représentant un événement fence en mémoire dans le langage de la machine par processus légers. On peut l'instancier avec tec_fencetau pour la machine par processus légers.*

```
Definition tec_fencetau e :=
match e with
| THmemc MEfence ⇒ True
| _ ⇒ False
end.
```

Blocage

— L'état actuel est bloquant pour la machine globale. L'abstraction de l'ordonnanceur est définie telle qu'une étape d'ordonnancement peut toujours s'exécuter. On a donc besoin de prendre en compte ses étapes d'ordonnancement dans la définition du blocage pour la machine globale :

```
Definition stuck s : Prop :=
 ∀ s' l s'', ∼ tau_sys _ l → tauschedstar s s' →  stepr ts s' l s'' → False.
```

— pour la machine par processus légers :

```
Definition stuck_t s :=
 ∀ s' l, stepr _ s l s' → False.
```

Erreur et blocage On a plusieurs prédicats représentant un cas d'erreur ou un blocage en fonction de la machine et du cas.

Une erreur peut être la prochaine étape exécutée par la machine globale ou la machine par processus légers :

```
Definition can_do_error s :=
 ∃ s', ∃ l, is_error _ l ∧  stepr ts s l s'.
```

Une erreur peut être atteinte après un nombre fini d'événements silencieux pour la machine globale :

```
Definition can_reach_error shms :=
  ∃ shms', tauscstar shms shms' ∧
            can_do_error shms'.
```

L'état actuel est bloquant ou émet directement une erreur pour la machine globale :

```
Definition stuck_or_error s := stuck s ∨  can_do_error s.
```

L'état actuel fait un nombre fini d'événements silencieux et de lecture avant d'émettre une erreur ou d'être bloqué pour la machine par processus légers.

```
Inductive ev_stuck_or_error_t s : Prop :=
 |  ev_stuck_or_error_stuck_t : stuck_t _ s → ev_stuck_or_error_t _ s
 | ev_stuck_or_error_error_t :
   ∀ l s', stepr _ s l s' → is_error _ l → ev_stuck_or_error_t _ s
 | ev_stuck_or_error_tau_t :
 ∀ l s', tau _ l → stepr _ s l s' →
       ev_stuck_or_error_t _ s' → ev_stuck_or_error_t _ s
 | ev_stuck_or_error_pseudotau_t :
 ∀ l s', pseudotau lbl l → stepr _ s l s' →
       (∀ l0 s'0, pseudotau _ l0 → stepr _ s l0 s'0 → ev_stuck_or_error_t _ s'0) →
       ev_stuck_or_error_t ts s.
```

L'état actuel fait un nombre fini d'événements silencieux avant d'émettre une erreur ou d'être bloqué pour la machine globale (on inclut le cas des événements d'ordonnancement) :

```
Inductive ev_stuck_or_error (s: St ts): Prop :=
 | ev_stuck_or_error_stuck:
   stuck s → ev_stuck_or_error s
 | ev_stuck_or_error_error:
   ∀ {l s'}, stepr ts s l s' → is_error lbl l → ev_stuck_or_error s
 | ev_stuck_or_error_tau:
   ∀ {s' l}, tso_or_tau lbl l →
     stepr ts s l s' →
     ev_stuck_or_error s' →
     ev_stuck_or_error s
 | ev_stuck_or_error_sched:
   ∀ {s' l}, tau_sys lbl l →
     stepr ts s l s' →
     ev_stuck_or_error s' →
     ev_stuck_or_error s
 | ev_stuck_or_error_pseudotau:
   ∀ {l s'},
     pseudotau lbl l →
     stepr ts s l s' →
     (∀ l s',
       pseudotau lbl l →
       stepr ts s l s' →
       ev_stuck_or_error s') →
```

```
ev_stuck_or_error s.
```

Mémoire séquentielle Dans tout nos développements, on réutilise les notions de mémoire utilisées dans Compcert que l'on verra pour des raisons de simplicité (dans ce mémoire) comme un modèle abstrait. En pratique, on a accès aux éléments sous-jacents mais on ne les utilise pas. Des définitions utiles à propos de ce modèle sont présentes en annexe B.

État de l'art

3

Sommaire

3.1 Compilation certifiée de langages séquentiels

Les premiers travaux sur la vérification de compilateurs ont presque cinquante ans. Dans [37], McCarthy et Painter s'intéressent à la correction de la compilation d'un petit langage d'expressions vers un petit langage machine les évaluant. Si la preuve est une preuve mathématique usuelle, son niveau de détails s'inscrit dans l'objectif de McCarthy [36]: « *to make it possible to use a computer to check proofs that compilers are correct* ». Depuis lors de nombreux travaux ont concerné la correction de compilateurs ou d'algorithmes de compilation, la bibliographie commentée de Dave [22] relève les travaux marquants jusqu'en 2003.

La première preuve d'une passe de compilation vérifiée avec un assistant de preuve semble être celle de Milner et Weyhrauch avec LCF en 1972 [38]. Le langage source est un petit langage impératif avec conditionnels et boucles et le langage cible un assembleur manipulant une pile. La preuve de correction de l'algorithme de compilation est une preuve que la sémantique dénotationnelle du programme cible correspond à la sémantique dénotationnelle du programme source. D'autres travaux récents se sont intéressés à la compilation vérifiée en se basant sur des sémantiques dénotationnelles : dans les deux cas les langages considérés ne sont pas des langages complets et la cible est un assembleur typé [6, 18]. Le traitement de langages plus réalistes se basent plutôt sur des sémantiques opérationnelles et l'inclusion de trace pour énoncer la correction du compilateur.

En 1972, Milner et Weyhrauch précisent bien que c'est l'algorithme de compilation qui est prouvé correct, non un compilateur particulier. Berghofer et Strecker [7] sont les premiers à utiliser les fonctionnalités d'extraction de l'assistant de preuve Isabelle pour obtenir un compilateur vérifié non-optimisant pour un sous-ensemble de Java. Le code extrait est interfacé avec du code non-vérifié pour l'analyse syntaxique puis la transformation du code assembleur en code-octet pour la JVM.

Verisoft [4] est un projet marquant qui avait pour ambition de fournir tout un environnement vérifié, incluant un processeur spécifique, un compilateur et un système d'exploitation. Au niveau du langage de programmation, *C0* est un sous-ensemble du langage C, sûr au niveau du typage et sans arithmétique de pointeurs. Le compilateur a été vérifié en Isabelle/HOL et ne comprend aucune passe d'optimisation.

C'est au milieu des années 2000, que le projet CompCert a commencé. L'objectif était d'implanter et de prouver la correction d'un compilateur *optimisant* pour un sous-ensemble large de C, avec l'assistant de preuve Coq [30, 31, 33, 13], dans un style qui permette l'extraction du compilateur en OCaml. Là encore le code extrait est intégré à du code non-vérifié pour les étapes initiale (analyse syntaxique) et finale (émission du code machine à partir de l'assembleur).

CompCert est organisé en de multiples passes et pour un langage machine cible donné (il peut générer du code assembleur pour trois architectures différentes: ARM, PowerPC et x86-32), il manipule dix langages : un léger sous-ensemble du langage C tel que défini par la norme, les langages Clight, C#minor, Cminor, CminorSel, RTL, LTL, Linear, Mach, et Asm (dépendant de l'architecture) qui sont des langages intermédiaires au compilateur et enfin l'assembleur de l'architecture considérée. La sémantique de C dans CompCert est surspécifiée par rapport à la norme du langage C : elle est déterministe, ainsi que tous les langages intermédiaires. Les preuves de correction des différentes passes de compilation peuvent ainsi être basées sur des simulations avant. Toutes les sémantiques sont des sémantiques à petits pas. CompCert est un compilateur optimisant. Il est notablement plus rapide que GCC sans optimisation, et de l'ordre de 20% plus lent que GCC au troisième niveau d'optimisation. CompCert est un développement logiciel conséquent : environ 130 000 lignes de Coq (la moitié de preuves).

Il est à noter que plusieurs travaux traitent du modèle mémoire de CompCert [12, 5, 11] : il s'agit ici essentiellement de différentes façons de représenter l'organisation de la mémoire, et non pas, comme dans le cas des modèles mé-

moire faibles, une description du fonctionnement des opérations mémoire. Les modèles les plus abstraits facilitent les preuves mais empêchent de capturer certaines fonctionnalités du langage C.

CompCert continue à évoluer, notamment par l'ajout d'optimisations supplémentaires, en particulier sous forme de *validateurs vérifiés* [52, 53, 44, 54]. Mais également en vérifiant des parties du compilateur qui était jusqu'alors non vérifiées, comme l'analyseur syntaxique [27]. Plus récemment [15], CompCert a été étendu pour traiter des nombres à virgule flottante.

De nombreuses autres équipes se basent sur CompCert pour leurs travaux, notamment pour la prise en compte du parallélisme.

3.2 Compilation certifiée et parallélisme

Un des premiers travaux concernant la compilation certifiée ciblant une architecture multi-cœurs avec un modèle mémoire faible est CompcertTSO défini dans [47, 48]. Le compilateur CompcertTSO réutilise les preuves de Compcert et conserve ses optimisations tout en fournissant des optimisations propres (liées au parallélisme). Le langage de plus haut-niveau est une extension du sous-ensemble de C de CompCert par l'ajout de création de processus légers, de barrière de synchronisation MEfence et d'instructions atomiques MErmw (read−modify−write).

Les auteurs étendent l'ensemble des langages considérés dans Compcert avec un modèle mémoire de type TSO (*Total Store Ordering*). Ils utilisent une machine mémoire et des notions d'« unbuffering » non déterministe pour simuler l'ordre donné par leur modèle mémoire. Les preuves utilisées dans Compcert ne sont plus suffisantes pour prouver la correction du compilateur. Néanmoins, il est démontré que les passes de compilation séquentielles impliquent des notions de simulation arrière qui, à leur tour, peuvent être combinées pour produire une simulation arrière globale qui entraîne la préservation de trace de leur système.

[56] ajoute des optimisations sur l'élimination de barrière de synchronisation qui sont non triviales et prouvées correctes dans CompcertTSO. D'autres extensions de Compcert tentent d'apporter une solution à la compilation de processus légers en parallèles notamment [8].

Un compilateur certifié pour un langage JAVA avec processus légers est proposé dans [34]. L'assistant Isabelle/HOL est utilisé pour formaliser et prouver la

correction de ce compilateur. Comme dans CompcertTSO, la preuve de correction du compilateur est obtenue par extension de la correction de la compilation séquentielle des processus légers du programme. En revanche, la consistance séquentielle est supposée bien que celle-ci ne soit pas vérifiée par les optimisations classiques réalisées par les compilateurs JAVA. Le compilateur proposé ne réalise toutefois aucune optimisation en lien avec la mémoire. Ce travail est basé sur un travail préliminaire, le compilateur JINJA [28] qui définit formellement la sémantique et la correction d'un compilateur pour la version séquentielle du langage. Comme pour CompcertTSO (et Compcert pour les version les plus récentes), une sémantique à petits pas est utilisée afin de modéliser l'entrelacement des processus légers.

3.3 Abstraction de modèles mémoire

Un modèle mémoire décrit comment un programme peut interagir avec la mémoire au cours de son exécution. Les implications du fonctionnement d'un processeur sur les opérations mémoire peuvent être extrêmement complexes. Il suffit de parcourir les centaines de pages des documentations techniques des processeurs actuels pour s'en convaincre. Les constructeurs de ces processeurs, qui souhaitent d'ailleurs dévoiler un minimum de détail sur la conception de leur matériel, restent très flous dans ces manuels techniques. Ce manque de précision peut conduire à une description erronée du comportement du processeur comme cela est démontré dans [41].

A l'opposé la description opérationnelle concrète du fonctionnement interne d'un processeur et donc de son modèle mémoire induit serait une tâche extrêmement complexe et fastidieuse. Il est donc nécessaire de disposer de descriptions formelles des modèles mémoire des processeurs suffisamment abstraite [39]. La littérature abondante sur les modèles mémoire propose différents niveaux d'abstraction de ces derniers Dans la première version de la spécification du modèle mémoire JAVA de Java [24], celui-ci était décrit par une modélisation semi-formelle d'une machine à base de caches afin de simuler les effets du modèle mémoire sur l'exécution des programmes. Il a été montré par la suite [43] que cette modélisation n'était pas suffisamment relâchée pour rendre compte de toutes les optimisations réalisées par les compilateurs JAVA existants. Cette découverte a donné lieu à de nombreux travaux visant à mieux rendre compte du comportement des programmes JAVA. En particulier, la spécification actuelle,

qui repose sur le JSR 133 [42] repose sur des contraintes d'ordres partiels sur les événements observables produits par l'exécution des programmes.

Des approches intermédiaires visant à capturer le comportement des programmes en présence de certaines optimisations ont également été proposées. Ces approches permettent de disposer d'une description opérationnelle et suffisamment abstraite du comportement des programmes comme par exemple dans [16]. Il reste malgré tout difficile de capturer de cette manière des comportements aussi variés que ceux des différentes architectures actuelles. Les modèles à base de contraintes d'ordre partiels restent donc très présent dans les différentes formalisation des modèles mémoire. Cette approche présente toutefois une difficulté technique importante car elle ne permet que de valider à posteriori un ensemble de comportements possibles.

Une autre approche, qui est celle de CompcertTSO que nous cherchons à généraliser, consiste à modéliser la mémoire sous la forme d'un oracle [25]. Finalement différentes approches visant à proposer un cadre uniforme pour l'étude des propriétés des programmes selon différents modèles mémoire ont été proposées [1, 57, 2].

4 Abstractions du modèle mémoire et de l'ordonnanceur

Sommaire

L'objectif de ce mémoire est d'exposer des moyens par lesquels on peut modulariser un compilateur certifié. Une première étape, et c'est l'objet de ce chapitre, est de séparer les différents éléments sémantiques d'un langage.

Cela permet d'isoler ses différentes parties pour pouvoir raisonner indépendemment sur chacune d'entre elle. C'est également une méthode facile pour ajouter ou modifier les langages.

Je m'inspire des méthodes utilisées par CompcertTSO pour créer différents sous systèmes synchronisables par les étiquettes des systèmes de transitions. J'utilise également les Module Type de Coq pour remplacer les états de l'exécution par des jeux d'hypothèses les décrivant. La représentation abstraite par un jeu d'hypothèses a pour but d'identifier les caractéristiques essentielles à la correction du compilateur. Créer une définition la plus faible possible conduit à pouvoir instancier un maximum de langages intéressants pour une étape de compilation donnée.

Dans ce chapitre concernant la définition des langages, je vais tout d'abord exposer la méthode que j'utilise pour modulariser la sémantique sur un exemple minimal. Puis, je vais décrire les limites qu'impose le modèle à travers la définition des étiquettes et de la machine globale. Enfin, je donnerai les abstractions des trois composants des languages : sémantique par processus légers, machine mémoire et machine ordonnanceur.

4.1 Exemple illustrant la méthode

Cette section traite d'un langage idéalisé de type RTL défini par une sémantique à petits pas (tel qu'il est défini dans Compcert). On entend comparer la définition sémantique d'une action mémoire définie dans Compcert à celle définie dans CompcertTSO dans le but d'expliquer celle-ci et de rendre naturelle l'extension de langage définie dans la suite de ce chapitre. Pour les besoins de l'explication, les détails de la définition des langages (notamment de RTL) sont omis. Une définition complète de ses langages est disponible dans [32].

4.1.1 Exemple d'approche par sémantique à petits pas

Rappel sur RTL RTL est un langage dans lequel les fonctions sont représentées par un graphe de flot de contrôle. La définition de Compcert contient trois types d'états suivant l'état de l'exécution : des états réguliers et des états particuliers pour les appels de fonctions et le retour de la valeur de ces fonctions. Dans cette partie, seule l'instruction *store* est considérée, tous les états sont donc supposés réguliers, et des éléments sont omis pour simplifier le propos. Un état régulier est composé des éléments suivants :

— une liste d'appels de fonctions Σ,

— un graphe de flot de contrôle de la fonction courante g,
— la position courante dans le graphe l,
— les valeurs des pseudo-registres R,
— l'état courant de la mémoire M.

Le système de transition correspondant à une étape d'exécution de l'instruction *store* est écrit de la façon suivante:

$$\frac{\begin{array}{c} eval(G, R(\vec{r})) = ptr(b, \delta) \\ g(l) = store(r, \vec{r}, l') \quad store(M, b, \delta, R(r)) = M' \end{array}}{G \vdash S(\Sigma, g, l, R, M) \xrightarrow{\tau}_{tid} \quad S(\Sigma, g, l', R, M')} \; store$$

L'emplacement l pointe vers une instruction *store* dans le graphe de flot de contrôle. Cette instruction a en argument r le registre contenant la valeur à ajouter en mémoire et $\vec{r}$, une liste de registres (pour cette instruction un seul) renvoyant l'emplacement auquel on mémorise la valeur de r. *eval* renvoie l'emplacement mémoire accédé (b, δ) avec b le bloc et δ le décalage. L'instruction de store dans les hypothèses signifie que la valeur $R(r)$ du registre r est enregistrée à l'emplacement (b, δ) pour donner un nouvel état mémoire M'. L'étape de transition est étiquetée par un événement silencieux τ. On a donc défini une transition d'écriture en mémoire en décrivant concrètement l'effet d'une telle instruction sur chacune des parties de l'état.

4.1.2 Exemple de spécification par étiquettes

Pour arriver à parler de mémoire partagée simplement, il est utile de pouvoir isoler la mémoire du reste de l'état d'exécution. L'idée est de changer la sémantique pour faire apparaître plusieurs états et plusieurs systèmes de transition étiquetés. Sans parler de parallélisme pour l'instant, on montre comment réécrire de façon équivalente l'étape de *store* précédente. La mémoire de l'état régulier est retirée et une étiquette est ajoutée sur la transition pour conserver l'information de *store* lorsque l'on communique avec la mémoire.

$$\frac{g(l) = store(r, \vec{r}, l') \qquad eval(G, R(\vec{r})) = ptr(b, \delta)}{G \vdash S(\Sigma, g, l, R) \xrightarrow{store\ b\ \delta\ R(r)}_{tid} \quad S(\Sigma, g, l', R)} \; Write$$

On ajoute une seconde entité sémantique pour la mémoire qui gère l'instruction de *store*:

$$\frac{store(M, b, \delta, v) = M'}{M \xrightarrow{store\ M\ \ b\ \delta\ v} M'} \; Writemem$$

Puis on ajoute un autre système de transitions étiquetées pour unifier les deux précédentes étapes en une:

$$\frac{s \xrightarrow{store\ b\ \delta\ v}_{tid} s' \qquad M \xrightarrow{store\ b\ \delta\ v} M'}{(s, M) \xrightarrow{\tau} (s', M')}\ Writeglob$$

Si on déplie les définitions des états s et M, on retrouve la définition de la section 4.1.1. Ces systèmes représentent le même langage.

4.1.3 Généralisation

On pense pouvoir adapter cette méthode à tout langage défini par un système de transitions étiquetées, considérant les effets mémoires comme étant instantanés. On peut donc faire correspondre un effet mémoire à une étape de réduction du langage. Cela est possible pour toute action, tout effet de bord qu'une étape d'exécution peut avoir. On peut utiliser ce point de vue comme base pour toutes les étapes communes à chaque langage intermédiaire d'un compilateur. Ces étapes ne sont alors plus définies que par des étiquettes et une machine permettant de les synchroniser : cela permet également d'abstraire le modèle mémoire et de potentiellement l'instancier de plusieurs façons différentes. Dans la suite, on étend cette notion à un langage avec plusieurs processus légers et un ordonnanceur permettant leur gestion.

Contexte avec ordonnanceur Une notion d'ordonnanceur est en effet possible : il suffit de créer un nouveau système de transitions étiquetées contenant des étiquettes autorisant un processus léger à s'exécuter. L'exécution du système de transitions étiquetées global est conditionnée par ces étiquettes d'ordonnancement. Par exemple, un ordonnanceur pourrait contenir une file d'attente d'exécution des processus légers permettant de stopper ou de restaurer différents processus légers en fonction de primitives de synchronisation lancées par le programme. Pour cela, on pose l'hypothèse que le système est de haut-niveau et qu'il permet d'exécuter chacun des processus légers indépendamment et d'intervertir les processus légers s'exécutant sans changer l'état courant de ces processus légers. Les processus légers font des accès concurrents à une mémoire globale.

On se place dans un langage idéalisé et dans le cas d'une instruction de type *store*. Cette restriction est suffisante pour l'explication du fonctionnement des

systèmes de transition étiquetés. Il faudra adapter les machines de façon cohérente pour obtenir le fonctionnement pour les autres instructions. Ceci sera détaillé dans la section suivante. Notre langage se compose d'une partie langage (C, assembleur, langage intermédiaire ...), d'une partie modèle mémoire, d'une partie ordonnanceur et d'une machine globale liant les différentes sous-parties. Quel que soit le langage considéré, une définition d'une étape dans la sémantique par processus léger, avec l, un pointeur vers la prochaine instruction à exécuter, $s[l]$ l'instruction en question (dans notre cas l'écriture à l'adresse p de la valeur v), et s' l'état d'arrivée qui ne change que par la position du pointeur vers l'instruction courante (uniquement dans ce cas, et pour l'instruction $l+1$) s'écrit :

$$\frac{s[l] = \text{store p v}}{s \xrightarrow{wr\ tid\ p\ v}_{tid} s'}\ Write$$

Dans la machine mémoire, on suppose l'existence d'un prédicat *allocated* qui comprend le fait que p fait référence à un emplacement mémoire existant, préalablement alloué et utilisable. On a $m' = m_{tid}[p \leftarrow v]$, c'est à dire l'état mémoire dans lequel on a changé la valeur contenu à l'emplacement p pour le processus léger tid. L'étape est :

$$\frac{allocated\ \ \text{p}}{m \xrightarrow{mem\ tid\ wr\ p\ v} m'}\ MeWrite$$

Dans la machine ordonnanceur, on cherche à gérer la possibilité d'exécution de processus légers. Pour faire simple, on représente l'état par une file d'attente dans laquelle l'identifiant de processus léger de tête peut s'exécuter. Ici, le processus léger tid est en tête et donc peut faire un pas d'exécution. Plusieurs fonctionnement de l'ordonnanceur sont possible. Ici, l'identifiant de processus léger revient à la fin de la file d'attente après une étape :

$$\frac{sc :: tid = tid :: sc'}{sc \xrightarrow{Tstep\ tid} sc'}\ SchedStep$$

On combine ces étapes avec une machine globale qui nous permet d'abstraire

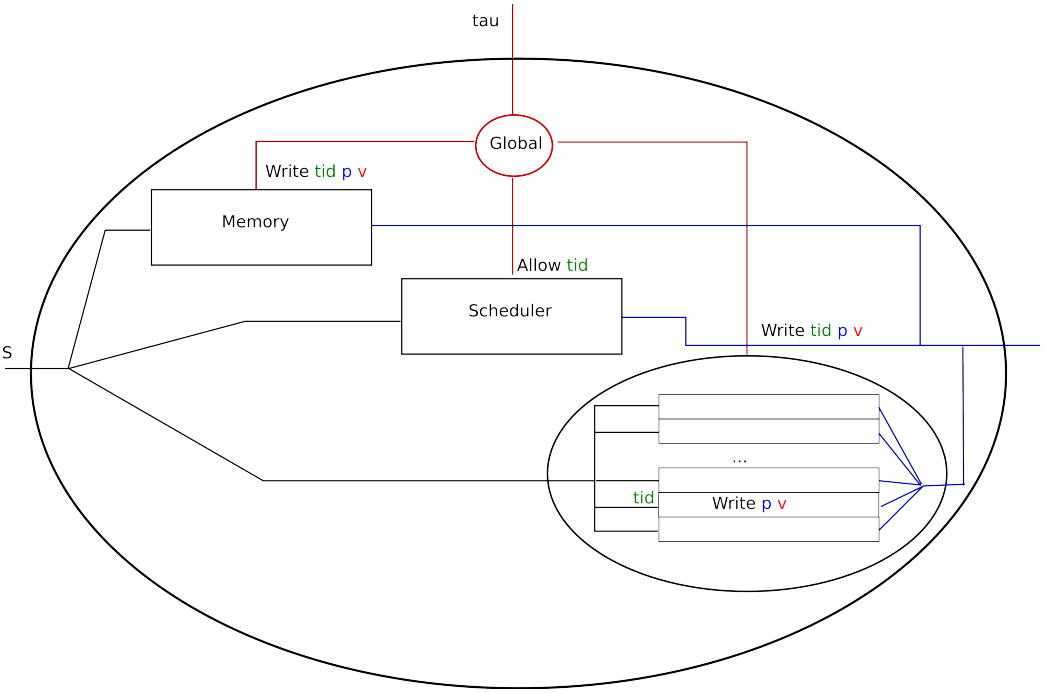


FIGURE 4.1 – *Machine globale*

les différentes étapes des machines et l'état de chacune d'entre elles :

$$\frac{sc \xrightarrow{Tstep\ tid} sc' \qquad s \xrightarrow{wr\ tid\ p\ v}_{tid} s' \qquad m \xrightarrow{mem\ tid\ wr\ p\ v} m'}{(s,m,sc) \xrightarrow{\tau_g} (s',m',sc')}\ Write$$

Les états s', m' et sc' sont donnés par les étapes *write* de leur système de transitions respectifs. Le fonctionnement de ce système de règles est schématisé à la figure 4.1 : un état de la machine globale S est scindé en plusieurs états qui font des étapes indépendamment dans les machines correspondantes et se synchronisent grâce à leurs étiquettes (en rouge).

On peut donner un équivalent de cette règle par dépliage des états et des règles :

$$\frac{sc :: tid = tid :: sc' \qquad s[l] = \text{write} \quad \text{allocated p}}{(s,m,sc) \xrightarrow{\tau_g} (s',m',sc')}\ Write$$

Dans notre formalisme, on peut voir que les seules réelles bornes à la modularité sont les étiquettes et leurs synchronisations (en dehors des problèmes liés à la compilation que l'on abordera dans la section 5.1) car ce sont les seuls paramètres fixés du système. Ce point de vue nous permet de changer différents éléments du langage sans redéfinir un langage complet. Par exemple, il est pos-

sible de changer le modèle mémoire en conservant les étiquettes présentes mais en changeant les règles de la partie mémoire du langage. Dans la section suivante, je donne des contraintes sur les différentes parties des langages en fixant les étiquettes que j'utilise.

4.2 Cadre spécifique

4.2.1 Étiquettes

On fixe une fois pour toute les différents jeux d'étiquettes que l'on utilise jusqu'à la fin du mémoire. Les étiquettes sont les seules interactions que l'on autorise entre les différentes parties de l'état global. Je les ai fixées de façon à pouvoir créer des langages comprenant une mémoire partagée et un ordonnanceur. Ces listes d'étiquettes sont exhaustives et leur définition est parfois arbitraire. Ces étiquettes sont grandement inspirées de celles que l'on trouve dans CompcertTSO.

Étiquettes d'événements mémoire séquentiels Certaines étiquettes de la machine mémoire et de la sémantique par processus léger prennent en argument un mem_event :

Définition 5 (Événements mémoires) *Une étiquette d'événement mémoire représente les différentes actions en mémoire (à accès non concurrent) possibles.*

```
Inductive mem_event :=
 | MEwrite (p: pointer) (chunk: memory_chunk) (v: val)
 | MEread  (p: pointer) (chunk: memory_chunk) (v: val)
 | MEalloc (p: pointer) (size: int) (k: mobject_kind)
 | MEfree  (p: pointer) (k: mobject_kind)
 | MEfence
 | MErmw   (p: pointer) (chunk: memory_chunk) (v: val) (instr: rmw_instr).
```

On note que l'on considère une barrière de synchronisation (MEfence) comme un événement mémoire pour des raisons de compatibilité avec CompcertTSO (il sert à imposer la propagation des événements mémoire dans TSO). MErmw permet une instruction atomique de type read−modify−write.

Étiquettes de processus léger Une étiquette de la sémantique par processus léger est définie par l'inductif THevent [1] de la façon suivante :

```
Inductive THevent :=
| THtau
| THmem (m: mem_event)
| THsyscall (sid: ident)
  (l: list (pointer * memory_chunk + val)) (* Liste d'arguments de l'appel *)
  (ltid: list thread_id) (* Liste d'identifiants de processus legers cree par l'appel *)
  (lv: list val) (* Liste de valeurs prises par les arguments de l'appel *)
| THsyscallfail (sid: ident)
  (l: list (pointer * memory_chunk + val)) (* Liste d'arguments de l'appel *)
  (ltid: list thread_id) (* Liste d'identifiant de processus legers cree par l'appel *)
  (lv: list val) (* Liste de valeurs prises par les arguments de l'appel *)
| THsysreturn (ty: typ) (v: eventval)
| THfail
| THoutofmem.
```

- THtau est un événement silencieux,
- THmem est un événement mémoire mem_event qui sont des actions mémoires uniques: *read*, *write*, *alloc*, *free*, *etc*,
- THsyscall représente un appel de fonction externe au langage (appel système),
- THsyscallfail correspond à l'échec d'un appel de fonction, par exemple si le type des arguments ne correspond pas à la signature de la fonction,
- THsysreturn représente le retour d'un appel système,
- THfail est une erreur dans l'exécution d'un processus léger,
- THoutofmem est un manque de mémoire dans un processus léger.

Il est possible d'ajouter des identifiants de processus légers représentant le processus léger par lequel l'étiquette est générée. On a besoin de ses identifiants mais pour des raisons techniques des développements Coq, on retrouve ces identifiants dans la définition des système de transitions étiquetées de la sémantique par processus légers (voir section 4.2.2).

Étiquettes de la machine mémoire Une étiquette d'événement de machine mémoire est définie de la façon suivante :

```
Inductive MMevent :=
| MMtau
```

1. Les définitions Coq grisées sont des liens vers les fichiers qui les contiennent.

```
| MMmem      (tid: thread_id) (m: mem_event)
| MMmemfail  (tid: thread_id) (m: mem_event)
| MMsys      (tid: thread_id) (id: positive) (realargs: list (pointer * memory_chunk + val))
  (args: list val) (ltid : list thread_id)
| MMsysfail  (tid: thread_id) (id: positive) (realargs: list (pointer * memory_chunk + val))
  (args: list val) (ltid: list thread_id)
| MMoutofmem (tid: thread_id) (i: int) (k: mobject_kind).
```

La machine mémoire, comme dans CompcertTSO, permet la gestion d'accès concurrents à une mémoire globale. Ici, on n'utilise pas directement de buffer mais on conserve une étiquettes τ de propagation des effets mémoire aux autres processus légers.

- MMtau est un événement silencieux. Par exemple, dans le modèle TSO, des événements sémantiques correspondent à des passages d'événements mémoire locaux à la mémoire globale ce qui permet de représenter le modèle mémoire relâché.
- MMmem est un événement mémoire classique: *read, write, alloc, free, etc*.
- MMmemfail est un événement mémoire classique qui a échoué (par exemple, libération d'un emplacement non alloué).
- MMsys est une synchronisation avec un appel de fonction externe. Cette étiquette permet entre autre de vérifier que les valeurs lues correspondent bien aux valeurs en mémoire (pour un langage bas-niveau qui place les arguments des appels de fonction en mémoire par exemple).
- MMsysfail indique l'échec d'un événement système mémoire. Par exemple, un échec pour charger une valeur faisant partie des arguments de l'appel système,
- MMoutofmem indique une mémoire insuffisante. Cette étiquette est présente pour d'éventuels futurs développement : actuellement, la mémoire est supposée infinie.

Étiquettes de la machine ordonnanceur Une étiquette d'événement de machine ordonnanceur est censée indiquer l'autorisation pour un processus léger de s'exécuter :

```
Inductive SCevent :=
 | SCsys: positive → thread_id → list thread_id → list val → SCevent
 | SCstep: thread_id → SCevent
 | SCtau: SCevent
 | SCfail: SCevent.
```

— SCsys correspond à une synchronisation avec un appel de fonction externe. Cela peut être une primitive de synchronisation (*lock, wait, notify*) qui a un effet sur l'état de l'ordonnanceur et donc sur les possibilités d'exécution des processus légers.
— SCstep indique qu'un processus léger est sur la pile d'exécution de l'ordonnanceur. Un événement non système peut s'exécuter sur le processus léger donné.
— SCtau indique un événement silencieux de l'ordonnanceur comme le changement d'un processus léger sur la pile d'exécution. Dans le cas d'une représentation de cet ordonnanceur par un ordonnanceur concret déterministe, on peut vouloir représenter des comportements qui rendent le langage inconsistant (à cause d'un ordonnanceur ne laissant jamais la sémantique par processus léger s'exécuter en ne faisant que des SCtau par exemple).
— SCfail indique une erreur interne à l'ordonnanceur.

Étiquettes de machine globale Les étiquettes d'événements de machine globale indiquent ce qui est visible de l'extérieur du système :

```
Inductive GMevent :=
 | GMvisible (ev: event)
 | GMtau
 | GMtausys (ev : tausys_event)
 | GMmmtau
 | GMoom.
```

— GMvisible traduit le fait qu'un appel système n'a pas été récupéré et est vraiment une entrées/sorties: appel ou retour de fonction. Par exemple, un appel à *printf*.
— GMtau est un événement silencieux du système. Soit un événement mémoire classique (*read, write, alloc, free, etc*) , soit un événement silencieux d'un processus léger, soit un appel système récupéré par l'ordonnanceur (pour une primitive de synchronisation).
— GMtausys est un événement τ particulier représentant les événements silencieux de l'ordonnanceur.
— GMmmtau représente un événement silencieux de la machine mémoire.
— GMoom est un cas d'erreur qui représente une mémoire insuffisante.

On sépare les différents événements τ suivant leur origine. En pratique, on pourrait les fusionner mais on souhaite garder la possibilité de les séparer si on

veut utiliser une trace d'exécution plus précise dans le futur. Des choix sont faits dans ces définitions qui sont volontairement orientées vers les instances disponibles dans les développements que l'on présente dans le chapitre 6. On pense néanmoins que d'autres implémentations utilisant ces définitions sont possibles. Ces définitions prennent un sens lorsqu'elles sont associées à une machine globale, elle aussi fixée, qui finit de contraindre le système.

4.2.2 Machine globale

La machine globale permet de synchroniser les différentes machines. Elle existe déjà dans CompcertTSO (voir section 3.2). Elle est définie par un état (le tuple des états mémoire, ordonnanceur et processus légers) et un système de transition. Cette machine n'est pas modulaire. Elle est fixée et est la même pour tout les langages. Ses étiquettes globales sont les actions réelles observables en dehors du système. Les effets de l'exécution des programmes sont définis grâce à ces événements.

Dans notre implémentation en Coq, la machine globale est représentée dans un Module GlobalMachine qui prend une machine mémoire et une machine ordonnanceur en paramètre. La sémantique par processus légers représentée par un Record SEM_G (voir section 4.3.2) pour des raisons de compatibilité avec CompcertTSO (en particulier les démonstrations séquentielles) est également un paramètre des définitions de GlobalMachine. Elle est caractérisée par un état state (tuple des états des différentes machines), par son système de transitions étiquetées step et par une fonction d'initialisation init. Le squelette de ce module que l'on détaille plus tard est :

```
Module Type GlobalMachine (MEMORY: Memory) (SCHED: Sched).
 Section GM.
  Variable Sem: SEM_G. (* Machine par processus legers *)
  Definition ST := SEM_G_ST Sem. (* Etat de la machine par processus legers *)
  Definition PRG := SEM_G_PRG Sem. (* Programme *)
  Definition state : Type := (MEMORY.state * ST * SCHED.state).
  Inductive step : state → GMevent → state → Prop := (* ... *).
  Definition init (PRG → list val → state → Prop) := (*... *) .
 End GM.
End GlobalMachine.
```

Le système de transitions étiquetées de cette machine step comporte un cas pour chaque événement prévu par les étiquettes fixes auxquelles correspondent des événements observables de la machine globale :

— à un événement mémoire (*lecture, écriture, allocation, désallocation, opération atomique*, correspond un événement silencieux GMtau (τ_g),
— à un événement silencieux de la mémoire correspond un événement GMmmtau ($tso\tau$),
— à un événement silencieux de l'ordonnanceur correspond un événement GMtausys ($sys\tau$),
— à un appel système (avec ou sans récupération par l'ordonnanceur) correspond un événement GMtau ou GMvisible ev (*visible*),
— à une erreur (mauvais appel système, erreur d'allocation ...) correspond un événement GMvisible fail (*visible fail*).

Dans la suite, nous regroupons les règles de ce système de transitions étiquetées pour une présentation plus claire.

Étapes mémoires

$$\frac{sc \xrightarrow{Tstep\ tid} sc' \qquad s \xrightarrow{TEmem\ ev}_{tid} s' \qquad m \xrightarrow{mem\ tid\ ev} m'}{(m,s,sc) \xrightarrow{\tau_g} (m',s',sc')}\ \textit{Ordinary memory}$$

$$\frac{sc \xrightarrow{Tstep\ tid} sc' \qquad s \xrightarrow{TEmem\ ev}_{tid} s' \qquad m \xrightarrow{memfail\ tid\ ev} m'}{(m,s,sc) \xrightarrow{fail} (m',s',sc')}\ \textit{Ordinary memory fail}$$

$$\frac{sc \xrightarrow{Tstep\ tid} sc' \qquad s \xrightarrow{TEmem\ (\mathbf{MEalloc}\ p\ n\ k)}_{tid} s' \qquad m \xrightarrow{moom\ tid\ n\ k} m'}{(m,s,sc) \xrightarrow{oom} (m',s',sc')}\ \textit{Out of memory}$$

$$\frac{s \xrightarrow{TEoom}_{tid} s' \qquad sc \xrightarrow{Tstep\ tid} sc'}{(m,s,sc) \xrightarrow{oom} (m,s',sc')}\ \textit{Thread out of memory}$$

$$\frac{m \xrightarrow{\tau_m} m'}{(m,s,sc) \xrightarrow{\tau_g} (m',s,sc)}\ \textit{Memory unbuffer}$$

— *Ordinary memory* est un accès mémoire: *allocation, désallocation, écriture, lecture, fence*.
— *Ordinary memory fail* est l'échec d'un accès mémoire. Les cas d'échecs dépendent de la définition du modèle mémoire. Par exemple, l'écriture à un endroit non alloué peut générer ce cas d'échec.
— *Thread out of memory* et *Out of memory* représentent des erreurs dues à un manque d'espace mémoire.
— *Memory unbuffer* est utilisé pour permettre une simulation compatible avec un modèle mémoire relaché(modèle mémoire TSO, par exemple).

Appels externes Pour les appels externes, on utilise une fonction qui permet de décider de l'étiquette globale. Cette fonction, decidefull est définie dans la partie ordonnanceur du langage et prend l'état de l'ordonnanceur en argument. Elle permet de changer l'étiquette globale en fonction de l'appel transféré à l'odonnanceur. Par exemple, un *lock* est censé se traduire par un τ_g, mais si le *lock* n'a pas été initialisé, il peut se traduire par un *visible fail*.

$$\frac{sc \xrightarrow{Tsys\ id\ tid\ ltid\ args} sc' \qquad s \xrightarrow{TEcall\ id\ argstso\ ltid\ args}_{tid} s' \qquad m \xrightarrow{msys\ tid\ id\ argstso\ args\ ltid} m'}{(m,s,sc) \xrightarrow{decidefull\ sc\ (Tsys\ id\ tid\ ltid\ args)} (m',s',sc')}\ Call$$

$$\frac{sc \xrightarrow{Tsys\ id\ tid\ ltid\ args} sc' \qquad s \xrightarrow{TEcall\ id\ largs\ ltid\ args}_{tid} s' \qquad m \xrightarrow{msysfail\ tid\ id\ largs\ args\ ltid} m'}{(m,s,sc) \xrightarrow{fail} (m,s',sc')}\ Call\ mem\ fail$$

$$\frac{sc \xrightarrow{Tsys\ id\ tid\ ltid\ args} sc' \qquad s \xrightarrow{TEcallfail\ id\ argstso\ ltid\ args}_{tid} s' \qquad m \xrightarrow{msys\ tid\ id\ argstso\ args\ ltid} m'}{(m,s,sc) \xrightarrow{fail} (m',s',sc'}\ Threadcall\ fail$$

$$\frac{s \xrightarrow{TEsysreturn\ ty\ v}_{tid} s' \qquad sc \xrightarrow{Tstep\ tid} sc'}{(m,s,sc) \xrightarrow{ret\ ty\ v} (m,s',sc')}\ Return$$

— *Call* est utilisé pour des appels à des fonctions externes à la sémantique par processus légers. Ces fonctions peuvent cependant avoir un sens pour la machine ordonnanceur. Par exemple, une fonction d'entrée/sortie n'a pas de sens pour la machine ordonnanceur mais une fonction de synchronisation en a un. C'est pourquoi on utilise la fonction decidefull qui permet de décider de l'étiquette globale en fonction de l'état de la machine ordonnanceur.
— *Call mem fail* est une erreur d'un appel de fonction dont la cause vient d'un problème mémoire. Par exemple, la lecture des arguments en mémoire peut générer cette erreur.
— *Threadcall fail* est une erreur dans l'appel de la fonction. Le nombre d'arguments donnés à la fonction peut ne pas correspondre à sa signature ou autre.
— *Return* est le retour d'un appel de fonction.

Exécution locale Certaines étapes sont locales à un processus léger (arithmétique par exemple). On les décrit ci-dessous :

$$\frac{s \xrightarrow{TEtau}_{tid} s' \qquad sc \xrightarrow{Tstep\ tid} sc'}{(m,s,sc) \xrightarrow{\tau_g} (m,s',sc')}\ \tau_n$$

$$\frac{s \xrightarrow{TEfail}_{tid} s' \qquad sc \xrightarrow{Tstep\ tid} sc'}{(m,s,sc) \xrightarrow{fail} (m,s',sc')}\ \textit{Thread fail}$$

$$\frac{s \nrightarrow l_{tid}\ s' \qquad sc \xrightarrow{Tstep\ tid} sc'}{(m,s,sc) \xrightarrow{fail} (m,s,sc)}\ \textit{Stuck thread}$$

— τ_n est une étape de l'exécution d'un processus léger qui se passe sans appels externes ou mémoire et sans erreur.
— *Thread fail* est un échec d'une commande propre au langage (pas un accès mémoire). Par exemple, une division par zéro.
— *Stuck thread* est un échec d'un processus léger: le fait de ne plus rien pouvoir exécuter alors qu'il en a la permission par la machine ordonnanceur est une erreur d'exécution.

Ordonnanceur

$$\frac{sc \xrightarrow{\tau_s} sc'}{(m,s,sc) \xrightarrow{\tau_g} (m,s,sc')} \textit{ Scheduler step}$$

$$\frac{sc \xrightarrow{fail} sc'}{(m,s,sc) \xrightarrow{fail} (m,st,sc)} \textit{ Scheduler fail}$$

— *Scheduler step* est une étape de l'exécution de la machine ordonnanceur qui est supposée pouvoir fonctionner indépendamment. Cette étape est interne à la machine ordonnanceur et permet d'avoir une temporalité de son comportement interne.

— *Scheduler fail* est une étape d'échec de la machine ordonnanceur.

La machine globale définit l'intégralité du système mais celui-ci peut être incorrect s'il ne respecte pas certaines propriétés simples de cohérence. Par exemple, une sémantique par processus légers qui ne respecte pas la propriété de réceptivité peut être considérée comme inconsistante. Elle n'est pas réceptive si pour un événement de lecture, la sémantique par processus légers fixe la valeur lue (voir définition 8). Dans ce cas, on n'utilise pas la machine mémoire et le langage est incohérent. Les sections suivantes exposent les hypothèses imposées pour une cohérence du système (hypothèses qui nous sont aussi nécessaires lors de la preuve de compilation).

4.3 Sémantique par processus légers

4.3.1 Définition

Nous reprenons les définitions et spécifications abstraites des sémantiques par processus légers de CompcertTSO et de la section 3.2. La sémantique d'un langage est abstraite par un Record comprenant :

— un type d'état,

— un système de transition (stepr ts),

— une fonction d'initialisation,

— plusieurs hypothèses relatives à l'état initial du programme, son environnement, les fonctions accessibles,

— des hypothèses relatives au parallélisme et à la cohérence de la machine globale (propriétés de réceptivité, détermination, et progrès présentes dans CompcertTSO).

Définition 6 *Deux événements sont de* samekind *s'ils sont égaux ou sont des événements de lecture d'un même emplacement avec le même type (ne se différenciant que par la valeur prise par l'emplacement). Cette propriété permet de représenter le fait que deux événements sont reliés sans avoir de contrainte sur les valeurs qui sont dans la machine mémoire. On la réutilise dans les propriétés de « réceptif » et de « déterminé ».*

```
Definition samekind (e1 e2 : event) : Prop :=
 match e1, e2 with
 | MEread p1 c1 v1, MEread p2 c2 v2 ⇒ p1 = p2 ∧ c1 = c2 ∧ type_constraint v1 v2
 | MErmw p1 c1 v1 f1, MErmw p2 c2 v2 f2 ⇒ p1 = p2 ∧ c1 = c2 ∧ f1 = f2 ∧
   type_constraint v1 v2
 | MEalloc p1 s1 k1, MEalloc p2 s2 k2 ⇒ s1 = s2 ∧ k1 = k2
 | Ereturn t1 rv1, Ereturn t2 rv2 ⇒ t1 = t2 ∧ type_constraint rv1 rv2
 | _, _ ⇒ e1 = e2
 end.
```

Définition 7 *Une sémantique par processus légers est « déterminée » si deux transitions partant d'un même état produisent un événement ne différant que par la valeur lue (si existante voir la définition 6) et s'ils produisent le même événement, on arrive dans le même état.*

```
Definition determinate : Prop :=
 ∀ s {s' s'' l l'}, stepr _ s l s' → stepr _ s l' s'' →
   (samekind lbl l l' ∧ (l = l' → s' = s'')).
```

Définition 8 *Une sémantique par processus légers est « réceptive » si toute lecture aurait pu être faite avec une valeur différente. Les valeurs contenues en mémoire sont gérées par la machine mémoire. Le fait que la sémantique par processus léger puisse fixer une valeur de lecture rend la machine mémoire inutile d'où la nécessité de cette hypothèse*

```
Definition receptive : Prop :=
 ∀ l {l' s s'},
   stepr ts s l s' →
   samekind lbl l' l →
   ∃ s'', stepr ts s l' s''.
```

Définition 9 *Une sémantique par processus légers a la propriété de « progrès » si, à tout moment, un processus léger peut soit continuer son exécution soit être bloqué.*

```
SEM_C_progress_dec :
```

```
Structure SEM_C := mkSEM_C
 {
  SEM_C_ST : Type ; (* type des etats *)
  SEM_C_GE : Type ; (* type de l'environnement *)
  SEM_C_PRG : Type ; (* type du programme *)
  (* initialisation de l'environnement: *)
  SEM_C_ge_init : SEM_C_PRG → SEM_C_GE → mem → Prop ;
  SEM_C_main_fn_id : SEM_C_PRG → ident ; (* pointeur vers la fonction main *)
  (* associe un identifiant de fonction a un pointeur vers celle−ci: *)
  SEM_C_find_symbol : SEM_C_GE → ident → option pointer ;
  (* systeme de transition: *)
  SEM_C_step : SEM_C_GE → SEM_C_ST → threadc_event → SEM_C_ST → Prop ;
  (* initialisation de l'etat: *)
  SEM_C_init : SEM_C_GE → pointer → list val → option SEM_C_ST ;
  (* proprietes: *)
  SEM_C_progress_dec :
   ∀ ge s, stuck_t (mklts threadc_labels (SEM_C_step ge)) s ∨
    (∃ l', ∃ s', SEM_C_step ge s l' s') ;
  SEM_C_receptive :
   ∀ ge, receptive (mklts threadc_labels (SEM_C_step ge));
  SEM_C_determinate :
   ∀ ge, determinate (mklts threadc_labels (SEM_C_step ge))
 }.
```

FIGURE 4.2 – *Enregistrement abstrayant la machine par processus léger*

```
∀ ge s, stuck_t (mklts threadc_labels (SEM_C_step ge)) s ∨
 (∃ l', ∃ s', SEM_C_step ge s l' s')
```

L'enregistrement abstrayant la sémantique par processus légers est dans la figure 4.2.

Les propriétés de réceptif, déterminé ou progrès sont essentielles pour une sémantique par processus légers. Sans ces propriétés, le système est incohérent.

En pratique, SEM_C est une sémantique par processus légers ne possédant qu'un seul processus léger. Dans CompcertTSO, les preuves séquentielles sont réutilisées pour être parallélisées. Ce Record est utilisé par CompcertTSO pour avoir une abstraction des langages séquentiels au niveau de la parallélisation.

4.3.2 Parallélisation des processus légers

Dans le but de simplifier la définition de la machine globale (section 4.2.2), nous avons créé un module pour gérer les détails de la gestion de plusieurs processus légers dans une même machine. Cette étape permet:

— de regrouper les différents processus légers dans le même objet,

```
Record SEM_G : Type := mkSEM_G
 { SEM_G_ST : Type; (* type des etats *)
   SEM_G_GE : Type; (* type de l'environnement *)
   SEM_G_PRG : Type; (* type du programme *)

   (* initialisation de l'environnement: *)
   SEM_G_ge_init : SEM_G_PRG → SEM_G_GE → mem → Prop;
   (* pointeur vers main: *)
   SEM_G_main_fn_id : SEM_G_PRG → ident;
   (* pointeur vers le corps des fonctions: *)
   SEM_G_find_symbol : SEM_G_GE → ident → option pointer;
   (* systeme de transition: *)
   SEM_G_step : SEM_G_GE → thread_id → SEM_G_ST → global_event →
     SEM_G_ST → Prop;
   (* initialisation de l'etat: *)
   SEM_G_init : SEM_G_GE → thread_id → pointer →
     list val → option SEM_G_ST;

   SEM_G_progress_dec : (* propriete de progres *)
   SEM_G_receptive : (* propriete de receptivite *)
   SEM_G_determinate : (* propriete de determine *)
   SEM_G_receptive_sys : (* propriete de receptivite pour les appels de fonctions *)
   SEM_G_indep : (* propriete d'independance *)
   SEM_G_indep_rev : (* propriete d'independance *)
   SEM_G_indep' : (* propriete d'independance *)
 }.
```

FIGURE 4.3 – *Enregistrement abstrayant la machine par processus légers parallélisée*

— de gérer les problèmes d'appels de fonction pour uniformiser les langages haut et bas-niveau. Le fait de passer les arguments en mémoire ou pas impose deux différents appels de fonction. SEM_G nous permet de cacher ce détail dans la machine globale,

— de gérer la création de processus légers qui se voit simplifiée dans la machine globale.

On définit donc SEM_G (figure 4.3) dont l'état représente un tuple d'états de SEM_C. La fonction de création de processus léger SEM_C_init n'existe plus pour la création de nouveaux processus légers, elle existe uniquement pour l'initialisation de la machine. Dans SEM_G, cette création est cachée dans le système de transition SEM_G_step. Les propriétés de SEM_G (figure 4.3) sont décrites par la suite.

Les définitions que l'on utilise sont des adaptations directes des définitions utilisées dans SEM_C :

Définition 10 (déterminé global) *La notion de déterminé global est exactement celle de déterminé.*

```
SEM_G_determinate : ∀ (tid : thread_id) (ge : SEM_G_GE),
                    determinate {| St := SEM_G_ST; stepr := SEM_G_step ge tid |};
```

Définition 11 (réceptif global) *La notion de réceptif global est exactement la même que réceptif.*

```
SEM_G_receptive : ∀ (tid : thread_id) (ge : SEM_G_GE),
                      receptive_global {| St := SEM_G_ST; stepr := SEM_G_step ge tid |};
```

Définition 12 (samekind global) *La version* samekind *globale est très similaire à samekind. On est cependant obligé d'introduire un cas pour les appels de fonction car le système de transitions étiquetées gère la mise en mémoire des arguments d'une fonction dans les langages bas-niveau. Il est donc nécessaire d'avoir une notion de samekind sur les arguments des appels de fonctions qui sont parfois des lectures en mémoire.*

```
Definition gb_samekind (e1 e2 : global_event) : Prop :=
 match e1, e2 with
 | TEmem (MEread p1 c1 v1), TEmem (MEread p2 c2 v2) ⇒
   p1 = p2 ∧ c1 = c2 ∧ type_constraint v1 v2
 | TEmem (MErmw p1 c1 v1 f1), TEmem (MErmw p2 c2 v2 f2) ⇒
   p1 = p2 ∧ c1 = c2 ∧ f1 = f2 ∧ type_constraint v1 v2
 | TEmem (MEalloc p1 s1 k1), TEmem (MEalloc p2 s2 k2) ⇒ s1 = s2 ∧ k1 = k2
 | TEsysreturn t1 rv1, TEsysreturn t2 rv2 ⇒ t1 = t2 ∧ type_constraint rv1 rv2
 | TEsyscall id1 argstso1 ltid1 args1, TEsyscall id2 argstso2 ltid2 args2 ⇒
   id1 = id2 ∧ argstso1 = argstso2
 | TEsyscall id1 argstso1 ltid1 args1, TEsyscallfail id2 argstso2 ltid2 args2 ⇒
   id1 = id2 ∧ argstso1 = argstso2
 | TEsyscallfail id1 argstso1 ltid1 args1, TEsyscall id2 argstso2 ltid2 args2 ⇒
   id1 = id2 ∧ argstso1 = argstso2
 | TEsyscallfail id1 argstso1 ltid1 args1, TEsyscallfail id2 argstso2 ltid2 args2 ⇒
   id1 = id2 ∧ argstso1 = argstso2
 | _, _ ⇒ e1 = e2
 end.
```

Définition 13 (appel réceptif) *En raison de l'abstraction des appels de fonctions, on introduit une nouvelle notion de réceptif sur les appels de fonction que l'on appelle appel réceptif.*

```
Definition receptive_g_sys (ts : lts global_labels) : Prop :=
∀ (l l' : global_event) (s s' : St ts), te_sys l → stepr ts s l s' →
 gb_samekind l' l → (∃ s'' : St ts, stepr ts s l' s'') ∨   (∃ s'', stepr ts s (opp_sys l') s'') .
```

où te_sys est une propriété qui caractérise une étiquette d'appel système :

```
Definition te_sys (ev: global_event) : Prop :=
  match ev with
    | TEsyscall _ _ _ _ ⇒ True
    | TEsyscallfail _ _ _ _ ⇒ True
    | _ ⇒ False
  end.
```

Définition 14 (indépendant) *Un système de transitions étiquetées est dit indépendant si l'événement silencieux généré par un processus léger n'a pas d'influence sur l'exécution des autres processus légers. Avec taustep défini dans la section 2.4.*

```
SEM_G_indep: ∀ ge tid l (s s1 s': SEM_G_ST), SEM_G_step ge tid s TEtau s1 →
∀ tid', tid <> tid' → taustep_t (_ tid') s l s' →
  ∃ s1', taustep_t (_ tid') s1 l s1';
```

Définition 15 (indépendance inversée) *Une autre notion d'indépendance est nécessaire: l'indépendance inversée.*

Les différents processus légers sont indépendants, si on peut générer un événement ev sur le processus léger tid, on peut toujours le générer après avoir fait un événement τ_n sur le processus léger tid'.

```
SEM_G_indep_rev: ∀ ge tid l (s s1 s': SEM_G_ST), SEM_G_step ge tid s TEtau s1 →
∀ tid', tid <> tid' → taustep_t (_ tid') s1 l s' →
  ∃ s1', taustep_t (_ tid') s l s1';
```

Nous avons créé une méthode standard pour passer de SEM_C à SEM_G. Elle dépend d'un module Parsing_fun qui a donc aussi une influence sur la sémantique globale.

4.3.3 Passage automatique à une sémantique par processus légers parallèlisée

Cette section décrit une méthode pour passer d'une définition locale de la sémantique par processus léger SEM_C à une définition parallélisée SEM_G. Cela permet de ne pas gérer les détails de la création de processus légers dans la

sémantique par processus légers au niveau d'abstraction au dessus de la sémantique par processus léger parallélisée. On a besoin de ce passage pour simplifier l'écriture de la machine globale qui ne contient plus les détails de la création de nouveaux processus légers dans la sémantique par processus légers (ajout d'un état pour un nouveau processus léger, réussite de l'initialisation). La suite de cette section contient la définition d'une fonction de passage Sem: SEM_C → SEM_G. Les états de la sémantique par processus légers parallélisée sont définis comme le tuple des états de la sémantique par processus légers pour chaque processus léger.

En Coq, on utilise la structure PTree.t SEM_C_ST qui a l'avantage de déjà exister dans Compcert et qui nous permet de représenter des associations finis d'éléments. Une structure abstraite (cette partie de la preuve n'est pas exécutable) aurait été préférable mais les bibliothèques sur les PTree existantes sont abstraites et suffisantes pour notre utilisation. La modification du système de transition étiqueté nécessite l'introduction d'une fonction de création de processus légers qui prend une liste d'arguments, un identifiant d'appel système et l'état de la sémantique par processus légers parallélisée et renvoie l'état après création des processus légers par l'appel système.

```
creation: SEM_C_GE Sem → list val → ident → list thread_id →
 PTree.t Sem.(SEM_C_ST) → option (PTree.t Sem.(SEM_C_ST)).
```

Les deux sous-sections suivantes détaillent respectivement la fonction de création et la génération du système de transitions étiquetées.

Signature d'appels systèmes et création de processus légers

Dans cette partie, on montre comment on peut créer une fonction de création de processus léger renvoyant un nouvel état d'exécution à partir d'une fonction de parsing et de SEM_C_init. La fonction de parsing permet d'isoler les différents éléments de la liste d'arguments d'un appel système. La fonction SEM_C_init dépend ensuite directement de ces éléments. Ce point est puremenet technique. La fonction de création est générée à partir de cette fonction parsing abstraite : parsing: list val → ident → list thread_id → option (list (pointer * list val)). En pratique, on instancie cette fonction d'une seule façon (avec un seul appel système de création particulier) car on n'utilise pas d'applications plus complexes. On présente donc directement cette fonction qui permet d'isoler la liste d'arguments des appels systèmes à la création d'un processus léger.

```
Definition parsing (lv: list val) (id: ident) (ltid: list thread_id) : option (list (pointer * list val)):=
match id with
| 1%positive ⇒ match lv with
 | Vptr p :: q ⇒ match ltid with
  | tid :: nil ⇒ Some ((p, q) :: nil)
  | _ ⇒ None
 end
 | _ ⇒ None
 end
| _ ⇒ None
end.
```

On combine parsing avec SEM_C_init pour obtenir la fonction de creation associée aux appels systèmes. Celle-ci prend l'état du système, un identifiant d'appel système et la liste des valeurs qui lui sont passées et renvoie le nouvel état du système s'il n'y a aucune erreur:

```
Fixpoint aux_fun (l: list (pointer * list val)) ltid (s: PTree.t Sem.(SEM_C_ST)) :=
match ltid, l with
| nil, nil ⇒ Some s
| (tid :: ltid'), (p, args) :: l' ⇒
 match (SEM_C_init Sem p args) with
 | None ⇒ None
 | Some s' ⇒ aux_fun l' ltid' (s ! tid <- s')
 end
| _, _ ⇒ None
end.

Definition creation sge lvor id ltid s :=
match (parsing lvor id ltid) with
| None ⇒ None
| Some l ⇒ aux_fun l ltid s
end.
```

Génération de sémantique parallélisée

Pour effectuer cette parallélisation, l'étape principale consiste à créer la sémantique de transition qui consiste à faire passer les événements de la sémantique par processus léger au niveau parallèle directement sauf pour les appels impliquant la création de processus légers. On part d'hypothèses générales sur

creation (qui sont vérifiées par la fonction présentée dans le paragraphe précédent) pour créer la fonction de passage Sem_g :

```
Variable Sem : SEM_C.

Variable creation: SEM_C_GE Sem → list val → ident → list thread_id →
 PTree.t Sem.(SEM_C_ST) → option (PTree.t Sem.(SEM_C_ST)).

Hypothesis creation_dec: ∀ ge fn lvor s,
 (∀ ltid, creation ge lvor fn ltid s = None) ∨
 (∃ s', ∃ ltid, creation ge lvor fn ltid s = Some s').

Hypothesis inv_creation_some: ∀ sge lvor id s s' tid ltid news,
 creation sge lvor id ltid  s = Some news →
 ∃ news', creation sge lvor id ltid (s ! tid <− s') = Some news'.

Hypothesis inv_creation_none: ∀ sge lvor id s s' ltid tid,
 creation sge lvor id ltid s = None →
 creation sge lvor id ltid (s ! tid <− s') = None.
```

Le nouveau système de transitions étiquetées parallélisé réutilise celui de SEM_C : on différencie les événements d'appels systèmes (propriété sys_eventc) des autres pour permettre la création de processus légers dans ces appels systèmes. Les événements systèmes sont distingués par deux cas : la création de processus léger est un succès ou un échec. Les autres événements sont inchangés. On utilise des fonctions qui extraient des information des événements d'appels de fonctions : arguments qui renvoient les arguments de l'appel, sys_id qui renvoie l'identifiant de la fonction.

```
Inductive Sem_g (sge: Sem.(SEM_C_GE)) (tid : thread_id)
 : PTree.t Sem.(SEM_C_ST) → global_event → PTree.t Sem.(SEM_C_ST) → Prop :=
(* Pas d'appels de fonction *)
| Not_event: ∀ st st' s l s' (H: ∼ sys_eventc l), Sem.(SEM_C_step) sge s l s' →
 st ! tid = Some s → st' = st ! tid <− s' →
 Sem_g sge tid st (thread_to_global_event l H) st'

(* Succes d'appels de creation de thread (ou pas d'appels) *)
| Sys_event: ∀ st st' st'' s l s' lvmem sid ltid lvor (H: sys_eventc l),
 Sem.(SEM_C_step) sge s l s' → st ! tid = Some s → st' = st ! tid <− s' →
 arguments l H = lvmem → sys_id l H = sid → creation sge lvor sid ltid st' = Some st'' →
 Sem_g sge tid st (TEsyscall sid lvmem ltid lvor) st''
```

```
(* Echec d'appels de creation de thread *)
| Sys_event_fail: ∀ st st' s l s' lvmem ltid sid lvor (H: sys_eventc l),
 Sem.(SEM_C_step) sge s l s' → st ! tid = Some s → st' = st ! tid <- s' →
 arguments l H = lvmem → sys_id l H = sid → creation sge lvor sid ltid st' = None →
 Sem_g sge tid st (TEsyscallfail sid lvmem ltid lvor) st'.
```

On génère ensuite réellement SEM_G à partir de SEM_C grâce à la définition par qui utilise des lemmes annexes pour les différentes propriétés de SEM_G :

- par_receptive et par_receptive_sys: possibilité pour deux événements similaire (Lecture d'un emplacement mémoire avec deux valeurs différentes par exemple) de s'exécuter depuis un état,
- par_determinate: depuis un état donné, les possibilités d'exécution génèrent forcément des étiquettes similaires et si celles-ci sont identiques, on arrive dans un état identique
- par_indep, par_indep_rev et par_indep': indépendance de l'exécution des événements de différents processus légers.

Démontrer l'intégralité des propriétés de SEM_G à partir de SEM_c et de creation_fun est naturel.

```
Program Definition par : SEM_G :=
 mkSEM_G (PTree.t (Sem.(SEM_C_ST)))
  (Sem.(SEM_C_GE))
  (Sem.(SEM_C_PRG))
  (Sem.(SEM_C_ge_init))
  (Sem.(SEM_C_main_fn_id))
  (Sem.(SEM_C_find_symbol))
  Sem_g
  _ _
  par_receptive
  par_determinate
  par_receptive_sys
  par_indep
  par_indep_rev
  par_indep'.
```

La dernière étape consiste à prouver que les propriétés de correction de la compilation peuvent être adaptée à la version parallélisée en utilisant cette méthode automatique de génération de SEM_G. On va tout d'abord regrouper un SEM_G source et cible créé par la fonction par dans un module Paral. Dans les preuves en Coq, on a des hypothèses sur les relations entre les SEM_C source

et cible, on a donc besoin de montrer que ces hypothèses sont préservées par la fonction par: SEM_C → SEM_G.

```
Module Type Paral (Import Src_Tgt: Lan_def_c) (Cs Ct: Creation_fun)
 (Import C: Creation Ct Cs) <: Lan_def_g.

Definition Src := par Src_Tgt.Src (creation Src Cs.creation) (creation_dec Src Cs.creation)
 (creation_inv_some Src Cs.creation) (creation_inv_none Src Cs.creation).

Definition Tgt := par Src_Tgt.Tgt (creation Tgt Ct.creation) (creation_dec Tgt Ct.creation)
 (creation_inv_some Tgt Ct.creation) (creation_inv_none Tgt Ct.creation) .

End Paral.
```

En pratique, on montre ensuite que l'on peut instancier le module de propriétés parallélisées de compilation (Global_par) à partir d'un module séquentiel (Correct_per_thread) mais que l'on ne détaille pas plus avant dans ce mémoire. La difficulté consiste à montrer que les relations entre états (de backward_simulation) sont conservées : en pratique on étend naturellement la relation entre états en imposant que deux états « parallèles » sont reliés si et seulement si leurs états par processus légers sont reliés par la relation processus léger à processus léger.

On démontre facilement mais laborieusement les propriétés nécessaires à l'obtention du module de compilation parallélisé concluant la méthode de parallélisation.

4.4 Modèle mémoire abstrait

Cette section est destiné à montrer comment abstraire la partie machine mémoire du langage et les hypothèses que l'on associe à la mémoire. On décompose cela en plusieurs parties: hypothèses de base sur une mémoire, hypothèses de cohérence, relations entre modèles mémoire source et cible, et simulation arrière. Chaque sous-section suivante contient une description d'une partie des hypothèses placée sur la mémoire.

Base Afin d'abstraire complètement l'état de la mémoire, on créée un Module Type qui abstrait cet état et n'y fait référence que par des propriétés. La base du Module Type du modèle mémoire peut donc être vu comme:

— un type d'état mémoire,

— un système de transitions étiquetées,
— une fonction d'initialisation.

Le squelette de base du module est le suivant. On définit les hypothèses ajoutées par la suite.

```
Module Type Memory.
(** Definition de l'etat memoire *)
Variable mem_state: Type.
(** Definition des transitions memoire *)
Variable mem_step: tso_state → tso_event → tso_state → Prop.
(** Fonction d'initialisation *)
Variable mem_init: mem → tso_state → Prop.
(* ... *)
End Memory.
```

Hypothèses de cohérence Cette définition de base du modèle mémoire s'accompagne d'un ensemble d'hypothèses spécifiant un « bon comportement » du modèle mémoire. Ces hypothèses sont nécessaires à la preuve de correction du compilateur mais se veulent suffisamment faibles pour pouvoir être instanciées par un modèle mémoire faible (par exemple le modèle mémoire TSO) ou un modèle mémoire séquentiellement consistant. On a besoin:

— d'une représentation de la mémoire perçue par un processeur per_proc_memory (sorte d'abstraction du système de buffer de TSO),
— d'hypothèses de cohérences sur per_proc_memory. Un événement mémoire classique sur un processus léger donné a un effet immédiat mais pas sur les autres processus légers,
— d'un ensemble d'identifiants de processus léger ayant déjà été « activées », on ne réutilise jamais les mêmes identifiants,
— d'hypothèses assurant que les identifiants de processus légers ne sont jamais repris, tso_alive renvoie une liste d'identifiants de processus légers déjà utilisés car on veut éviter de réutiliser des identifiants même de processus légers ayant terminer leur exécution (pour des raisons de simplicité).

```
(** La machine est capable de fournir la memoire vu par chaque processeur *)
Parameter per_proc_memory: mem_state → thread_id → mem.

(** On suppose que chaque evenement de la machine sur un processeur pid
 est instantanement visible sur ce processeur *)
```

```
Parameter coherence_seq: ∀ (ev: mem_event) ttso ttso' pid,
 mem_step ttso (TSOmem pid ev) ttso' → operation_mem ev ttso ttso' pid.

(* Pour un thread qui ne s'execute pas, un evenement memoire d'un autre thread peut
etre retarde. Les evenement MMtau sont utilises pour propager les actions memoires *)
Parameter coherence_oth': ∀ (ev: mem_event) ttso ttso' pid pid' (Hrmw: ∼ mermw ev),
 pid <> pid' → mem_step ttso (TSOmem pid ev) ttso' →
 (per_proc_memory ttso pid' = per_proc_memory ttso' pid').

Parameter tso_alive: mem_state → list thread_id.

Parameter keep_alive: ∀ tso tso' ev,
 mem_step tso ev tso' → ∀ tid, In tid (tso_alive tso) → In tid (tso_alive tso').

Parameter step_alive: ∀ ttso ttso' tid ev,
 mem_step ttso (TSOmem tid ev) ttso' → In tid (tso_alive ttso').
```

On ajoute également des hypothèses de gestion du parallélisme. Elles permettent de fournir des informations sur les événements mémoire qui n'ont pas été propagés entre processus léger s. On renvoie le lecteur au code Coq pour des informations sur ces hypothèses.

Relations entre modèles mémoire Dans le cadre de la compilation de langages, on se doit de parler des relations entre langages source et cible. On a voulu pouvoir changer le modèle mémoire en fonction du niveau du langage : c'est possible mais on doit ajouter des hypothèses de cohérence abstraites entre modèle mémoire source et modèle mémoire cible. C'est l'objet de cette section.

Même si l'on ne change pas de modèle mémoire durant une étape de compilation, on doit fournir une relation de simulation entre les états du modèle mémoire source et celui du cible. Dans le cas général, avec un seul modèle mémoire, une telle relation de simulation ne peut pas être prouvée avec la spécification précédente du modèle mémoire car des hypothèses nécessaires à la compilation peuvent être plus forte que ces hypothèses. Et, plus généralement des hypothèses de cohérence entre les Module Type définissant la machine mémoire source et la machine mémoire cible sont nécessaires. Cela contraint, implicitement, fortement les modèles mémoire que l'on peut utiliser.

La relation entre modèles mémoire dépend en grande partie de la phase de compilation. Dans notre travail, on distingue trois relations distinctes. Les deux

premières correspondent à des phases de compilation qui n'ont pas ou peu d'influence sur la mémoire et pour lesquelles il suffira d'avoir une relation d'égalité entre les états sources et cibles. La troisième est plus complexe car la phase de compilation implique de grands changements en mémoire : en particulier, il y a une réallocation complète des emplacements mémoire.

Pour les deux premières, on définit le Module Type d'interactions entre modèles comprenant principalement :

— une relation mem_rel entre les états source et cible,
— une hypothèse de backward_simulation sur mem_rel,
— une hypothèse de correspondance entre les états initiaux.

La partie intéressante se situe dans la partie backward_simulation.

Simulation arrière en mémoire La backward_simulation dépend de deux paramètres qui nous permettent de généraliser des résultats dans la phase de preuve de correction. On se base sur un module abstrait Mem_io qui définit en particulier mldi et mldo qui sont des hypothèses de cohérences entre des événements similaires sources et cibles. Une des instances est la suivante :

```
Definition mldi := @eq tso_event.

Definition mldo (ev ev': tso_event) := False.
```

Dans ce contexte, l'hypothèse de backward_simulation (simplifiable en remplaçant mldi et mldo et en propageant) est:

```
Hypothesis valid_tso_bs: ∀ ttso ttso' stso ev,
  Ttgt.mem_step ttso ev ttso' → tso_rel stso ttso →
  ((∃ ev', Mio.mldo ev' ev ∧
      ∃ stso', tsotstep Tsrc.mem_step stso ev' stso' ∧  tso_rel stso' ttso')
  ∨  ∀ ev', Mio.mldi ev' ev →
     ∃ stso', tsotstep Tsrc.mem_step stso ev' stso' ∧  tso_rel stso' ttso').
```

En la simplifiant, on obtient une backward_simulation naturelle:

```
Hypothesis valid_tso_bs: ∀ ttso ttso' stso ev,
  Ttgt.mem_step ttso ev ttso' → tso_rel stso ttso →
     ∃ stso', tsotstep Tsrc.mem_step stso ev stso' ∧  tso_rel stso' ttso').
```

Le module Mem_io (mldi et mldo) sert à définir une relation entre les entrées/sorties mémoire (lecture, écriture) des différents niveaux. Le comportement attendu est de pouvoir effectuer une action de *read* tant que la valeur en mémoire est la valeur de la mémoire de la machine cible ou une valeur moins définie (au sens de Compcert).

Cas particulier Dans certains cas, les changements faits en mémoire sont assez importants et leur adaptation nécessite d'avoir recours à des propriétés plus exotiques. Ainsi on définit une backward_simulation:

— une relation mem_rel entre les états mémoire sources et cibles,
— une relation d'initialisation sur mem_rel,
— une hypothèse de backward_simulation très différenciée selon les événements émis,
— neaucoup d'hypothèses ad-hoc que l'on ne détaille pas dans ce mémoire.

4.5 Ordonnanceur abstrait

4.5.1 Définition

La machine ordonnanceur se définit de la même manière que le modèle mémoire. L'ordonnanceur doit pouvoir gérer l'exécution des processus légers, autoriser leur exécution ou contraindre leur ordre d'exécution. Les étiquettes de l'ordonnanceur ont été choisies en conséquence. Notamment, SCsys permet la communication d'appels systèmes à l'ordonnanceur et SCstep permet à l'ordonnanceur de contraindre l'exécution d'un processus léger plutôt qu'un autre. On va, comme pour la machine mémoire, abstraire l'ordonnanceur dans un Module Type contenant :

— le type des états de l'ordonnanceur sched,
— un système de transition scheduler_step,
— une fonction d'initialisation ext_init,
— une fonction globale. Elle prend une étiquette et l'état de l'ordonnanceur et décide si l'événement est propagé à la machine globale et de quelle façon (exemple : un événement d'ordonnanceur de verrouillage est changé en événement global interne),
— une liste de tous les identifiants de processus légers déjà utilisés.

```
(** * Module defining the external machine *)
Module Type Sched.
(** Type of scheduler state *)
Parameter sched: Type.
(** Ext machine definition *)
Parameter scheduler_step: sched → sevents → sched → Prop.
(** Creation of a scheduler state (and correctness of this creation) *)
```

```
Parameter ext_init: thread_id → sched → Prop.
Parameter decide_full: sched → sevents → fullevent.
(** We assume we can enumerate the id of thread already used
We want to be sure we dont reuse the same name *)
Parameter active_thread: sched → list thread_id.
(*...*)
End Ext.
```

On ajoute également des hypothèses de cohérence sur ce module. On a, par exemple, besoin que les processus légers actifs le restent toujours au cours de l'exécution et qu'ils s'ajoutent à la liste des processus légers actifs lorsqu'on les créé.

4.5.2 Relation entre ordonnanceurs

Comme pour le modèle mémoire, on a besoin d'une relation entre machines ordonnanceurs pour la preuve de correction de la compilation Ces relations sont basées sur l'idée de backward_simulation et vont potentiellement créer des contraintes sur les différentes machines ordonnanceurs utilisables.

On se base sur une relation entre états sched_rel pour laquelle on définit une simulation arrière à un pas valid_sched_bs, une initialisation sched_rel_init et une propriété de blocage valid_sched_stuck qui nous servent de base pour l'abstraction. Pour l'explication, et par abus de langage, on omet de parler des relations entre états dans les explications suivantes (les propriétés Coq sont présentes ci-après). La propriété valid_sched_bs impose qu'un événement de la machine ordonnanceur cible est simulé par la machine ordonnanceur source après un nombre fini d'événements internes ou que celles-ci va rentrer dans un état d'erreur. La propriété sched_rel_init associe un événement initial source à un événement initial cible. La propriété valid_sched_stuck traduit la possibilité pour la machine cible de simuler un événement de la machine source après un nombre fini d'étapes internes (ou d'étapes silencieuses d'autres processus léger). Cette propriété peut également se lire par un blocage de la machine source lorsque la machine cible est bloquée.

```
Module Type Ext_rel_1 (SrcE TgtE: Ext).

 Variable sched_rel: SrcE.sched → TgtE.sched → Prop.

 (** "backward simulation" for the scheduler *)
 Hypothesis valid_sched_bs: ∀ tsched tsched' ssched ev,
```

```
  TgtE.scheduler_step tsched ev tsched' → sched_rel ssched tsched →
  (∃ ssched'', ∃ ssched', @schedstar _ SrcE.scheduler_step ssched ssched' ∧
  ((SrcE.scheduler_step ssched' ev ssched'' ∧
  sched_rel ssched'' tsched') ∨  (ev <> TStau ∧
                                    SrcE.scheduler_step ssched' TSfail ssched''))).
(** Initialisation properties *)
Hypothesis sched_rel_init: ∀ tid tsched, TgtE.ext_init tid tsched →
  ∃ ssched, SrcE.ext_init tid ssched ∧  sched_rel ssched tsched.

Hypothesis valid_sched_stuck': ∀ ssched ssched' ssched'' tsched ev,
  sched_rel ssched tsched → @schedstarstep _ SrcE.scheduler_step ssched ssched' →
    SrcE.scheduler_step ssched' ev ssched'' →
    ∃ tsched', ∃ tsched'', @schedstarstep _ TgtE.scheduler_step tsched tsched' ∧
      TgtE.scheduler_step tsched' ev tsched''.

(* ... *)
```

On ajoute également une relation entre les fonctions decide_full qui permettent de gérer la visibilité des événements dans la machine globale.

```
Hypothesis decide_full_rel: ∀ ssched tsched sev,
  sched_rel ssched tsched →
    SrcE.decide_full ssched sev = TgtE.decide_full tsched sev.
```

Cette spécification constitue la base de raisonnement que l'on utilise dans les preuves du chapitre 5. Mais, ces hypothèses ne décrivent pas l'ordonnanceur de façon assez précise pour achever ces démonstrations. On a donc dû ajouter des hypothèses sur les relations entre machines ordonnanceurs (en vérifiant qu'elles soient instanciables) pour achever les preuves. Elles sont moins intuitives que celles énoncées précédemment. Les hypothèses step_access_rev et step_access_rev_tgt enrichissent les machines ordonnanceurs d'une hypothèse d'accessibilité : si l'ordonnanceur permet d'arriver à l'exécution d'un processus léger sans événement système (access), il le permet toujours après avoir laissé d'autres processus légers s'exécuter.

```
 Hypothesis step_access_rev: ∀ sched schedi tid,
   @schedstarstep _ SrcE.scheduler_step schedi sched →
     @access _ SrcE.scheduler_step tid sched →
       @access _ SrcE.scheduler_step tid schedi.

 Hypothesis step_access_rev_tgt: ∀ sched schedi ev (Hs: ev <> TStau),
   schedstarstep TgtE.scheduler_step sched schedi →
```

```
access' TgtE.scheduler_step schedi ev → access' TgtE.scheduler_step sched ev.
```

scheduler_block représente une notion de blocage comme valid_sched_stuck. Si la machine ordonnanceur cible ne peut plus avancer (avec des étapes internes) alors la machine source ne pourra plus avancer à partir d'un certain rang (ou renverra une erreur). Par exemple, le comportement de la machine ordonnanceur source ne pourra pas boucler indéfiniment et se traduire vers une machine ordonnanceur cible qui termine. On a besoin de préciser cette hypothèse pour SCtau car la définition de valid_sched_stuck est trop faible dans ce cas là.

```
Hypothesis scheduler_block:
 ∀ ssched tsched,
 sched_rel ssched tsched →
 (∀ tsched', ∼ TgtE.scheduler_step tsched TStau tsched') →
 ∃ ssched', schedstar SrcE.scheduler_step ssched ssched' ∧
 ( (∀ ssched'', ∼ SrcE.scheduler_step ssched' TStau ssched'') ∨
  ∃ ssched'', SrcE.scheduler_step ssched' TSfail ssched'').
```

step_succ' traduit l'idée que l'ordre des événements SCstep et SCtau n'importe pas tant que l'ordre des événements SCsys est respecté. Si on peut partir d'un état sched, faire des événements SCtau puis un événement ev pour arriver dans un état sched'' alors tout état schedi accessible depuis sched par l'exécution de différents processus légers peut atteindre un état schedf par un même événement ev. Et les états sched'' et schedf peuvent être reliés à un même état de la machine ordonnanceur cible tsched.

```
Hypothesis step_succ': ∀ sched sched' sched'' tsched ev,
 schedstar SrcE.scheduler_step sched sched' →
 SrcE.scheduler_step sched' ev sched'' →
 sched_rel sched'' tsched → ∀ schedi,
 schedstarstep SrcE.scheduler_step sched schedi →
 ∃ schedi', ∃ schedf,
  schedstar SrcE.scheduler_step schedi schedi' ∧
  SrcE.scheduler_step schedi' ev schedf ∧ sched_rel schedf tsched.
```

On ajoute également une hypothèse relative à la machine ordonnanceur source et permettant de gérer les étapes de compilation nécessitant l'utilisation des lessdef. Si un état de la machine ordonnanceur peut gérer un événement Tsys avec des arguments donnés, il peut le faire avec des arguments moins définis après un nombre fini d'étapes SCtau (ou émettre une erreur) et ce en conservant une notion de relation avec un état de la machine cible donnée.

```
Hypothesis sched_lessdef: ∀ sched sched' sched'' tsched tsched' id tid ltid args args0
```

```
(LD: Val.lessdef_list args0 args),
schedstar SrcE.scheduler_step sched sched' →
SrcE.scheduler_step sched' (Tsys id tid ltid args) sched'' →
sched_rel sched'' tsched' → sched_rel sched tsched →
∃ sched0', ∃ sched0'',
  schedstar SrcE.scheduler_step sched sched0' ∧
  SrcE.scheduler_step sched0' (Tsys id tid ltid args0) sched0'' ∧
  sched_rel sched0'' tsched' ∧  sched_rel sched0' tsched ∧
  (SrcE.decide_full sched (Tsys id tid ltid args) =
    SrcE.decide_full sched0' (Tsys id tid ltid args0) ∨
  SrcE.decide_full sched0' (Tsys id tid ltid args0) = Evisible Efail ).
```

Enfin, on a également eu besoin d'hypothèses fortes sur la relation sched_rel : un état se réduisant en produisant un événement SCtau ou SCstep doit être en relation avec les mêmes états par sched_rel.

```
(** Tstep or TStau should not have any influence on the relation because only sys
 event should be cared about *)
 Hypothesis rel_sched_silent_s: ∀ ssched ssched' tsched ev,
  tau_or_step ev → SrcE.scheduler_step ssched ev ssched' →
  (sched_rel ssched tsched <→ sched_rel ssched' tsched).

 Hypothesis rel_sched_silent_t: ∀ ssched tsched' tsched ev,
  tau_or_step ev → TgtE.scheduler_step tsched ev tsched' →
  (sched_rel ssched tsched <→ sched_rel ssched tsched').
```

Ces hypothèses fortes transparaissent déjà dans step_succ et sched_lessdef. En changeant les démonstrations qui nécessitent ces hypothèses, il parait probable que l'on arrive à les supprimer ou, au contraire à définir une relation plus simple. On verra dans la section 6.2.1 que l'on peut définir une spécification plus contraignante pour les machines ordonnanceurs. Cette spécification est une instance de celle qu'on vient de présenter. On peut l'utiliser dans les cas simples de machines ordonnanceurs comme on le verra dans la section 6.2.1.

Conclusion

Dans ce chapitre, nous avons utilisé une méthode de définition de sémantiques opérationnelles de langages par systèmes de transitions étiquetées synchronisés pour étendre l'approche de CompCertTSO en donnant le moyen de

parler de primitives de synchronisation et d'ordonnancement grâce à l'ajout d'un ordonnanceur.

Par souci de modularité, nous avons également abstrait ces différentes machines en les représentant par un Module Type de spécification. Le reste du développement est donc *parametré* par des modules de ces types. Nous pourrons ainsi présenter différentes instances de ces modules compatibles avec le compilateur dans le chapitre 6.

Correction de compilateur étendu 5

Sommaire

Le chapitre précédent présente la formalisation de sémantiques de langages parallèles, de façon modulaire, créés par la synchronisation d'une sémantique par processus léger, d'une machine mémoire et d'une machine ordonnanceur. Cette synchronisation est faite par les étiquettes des systèmes de transitions régissant l'exécution du langage.

Dans le présent chapitre, nous nous intéressons à la correction d'une passe de compilation entre deux langages définis de cette manière. Cette correction est établie par la préservation de la trace d'exécution générée par les étiquettes des machines globales : l'ensemble des traces d'exécution possibles de la machine cible est inclus dans l'ensemble des traces d'exécution possibles de la machine source. Pour prouver la correction d'un compilateur, il est plus aisé de manipuler une notion de simulation arrière dont l'énoncé repose sur une équivalence entre états des machines globales des langages sources et cibles. Dans un premier temps (section 5.1), nous présentons la propriété de simulation arrière et la relation d'équivalence sur les machines globales, puis dans un second temps (section 5.2), nous expliquons la preuve de conservation de trace.

5.1 Propriété générale de simulation arrière

Pour montrer la correction du compilateur, on se base sur des propriétés de simulation arrière à un pas séquentielles dont on prouve qu'elles impliquent

des simulation arrière à un pas parallèles. Seulement, ces simulations arrières à un pas dépendent fortement des passes de compilation considérées, il est donc nécessaire d'adapter ces définitions et preuves. En particulier, les passes de compilation se différencient fortement par les changements en mémoire qu'elles impliquent

Après avoir énoncé les propriétés de simulation arrière que nous utilisons, nous définissons une première version de l'équivalence d'états globaux. Dans la section 5.1.1 nous traitons du cas des passes de compilation à faible interaction mémoire et dans la section 5.1.2 de celles pour lesquelles l'interaction mémoire est plus forte.

Propriétés de la simulation arrière Dans cette section, on simplifie systématiquement les énoncés Coq de façon à ne pas perdre l'idée principale dans le détail des définitions et des lemmes (en particulier, les relations sur les environnements des programmes sont omises). On suppose que l'on dispose d'une machine globale source G_s et d'une machine globale cible G_t comprenant chacune une machine mémoire, une sémantique par processus légers et une machine ordonnanceur. On suppose également qu'il existe des relations mem_rel, thread_rel, sched_rel entre respectivement les états des machines source et cible qui respectent les propriétés données par les abstractions que l'on a donné au chapitre 4.

Les différentes propriétés que l'on veut obtenir sur ces relations et que l'on détaille dans la suite se séparent en trois types qui sont déjà présents dans CompcertTSO (sous une forme adaptée au contexte) :

— relation d'initialisation entre les états source et cible,
— relation de blocage entre des états bloqués,
— simulation arrière à un pas entre des états source et cibles reliés.

En pratique, on veut montrer la conservation de trace. Intuitivement, on a besoin de l'initialisation pour s'assurer qu'il existe un état initial source associé à un état initial cible. La relation de blocage est nécessaire pour s'assurer qu'à un état bloqué (trace finie) on fait correspondre un autre état bloqué (trace également finie). La relation de simulation arrière sert à faire correspondre les événements d'exécution entre les états correspondant. Ces éléments sont présentés et démontrés indépendemment.

Initialisation Si src_prog se compile vers tgt_prog (propriété match_prg) et que tstate est un état initial valide (propriété init) pour l'environnement cible donné

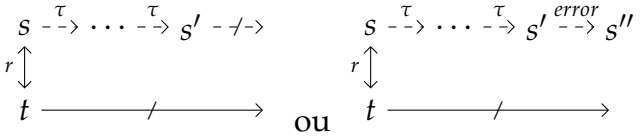
alors il existe un état initial sstate pour l'environnement source tel que les deux états sont reliés par la relation full_rel. En d'autres termes, pour tout état initial valide pour un programme compilé, il existe un état initial valide pour le programme source et les deux sont en relation.

```
Lemma init_related:
  ∀ src_prog tgt_prog args tstate,
    match_prg src_prog tgt_prog →
    init tgt_prog tstate →
    ∃ sstate,
      init src_prog args sstate ∧ full_rel sstate tstate.
```

Blocage La relation de blocage établit qu'à un état de la machine globale cible bloqué correspond forcément un état source qui peut se bloquer (ou retourner une erreur) après un nombre fini d'étapes silencieuses (étiquette GMtau). Le prédicat stuck s traduit que l'état s ne peut plus effectuer d'étape, tauscstar s s' que s peut effectuer un nombre fini d'étapes silencieuses pour arriver en s' et stuck_or_error s' que l'état s ne peut plus effectuer qu'une étape d'erreur.

```
Lemma blocked_sim:
  (∀ s t, stuck _ t → full_rel s t →
    ∃ s', tauscstar _ s s' ∧ stuck_or_error _ s').
```

Ce lemme peut être représenté par le diagramme suivant qui voit la traduction de deux cas. Soit, s ne peut plus se réduire soit s' peut se réduire par une étape d'erreur.



Simulation arrière à un pas La relation de simulation arrière à un pas permet de faire correspondre une étape de l'exécution du programme cible à une ou plusieurs étapes du programme source. L'idée est de simuler cette étape par une même étape dans le programme cible. En pratique, on distingue trois cas :

— Une étape se produit dans le programme cible et, après un nombre fini d'étapes silencieuses, une étape similaire se produit dans le programme source,

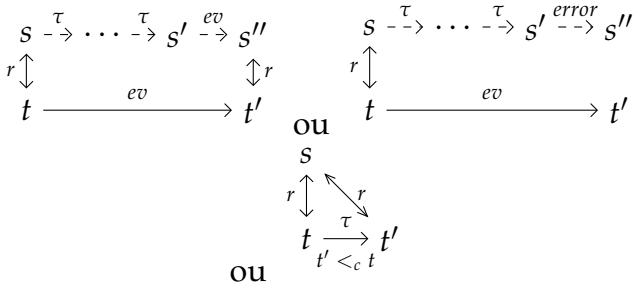
— une étape se produit dans le programme cible, l'état d'arrivée est inférieur à l'état d'origine pour la mesure donnée et l'état d'arrivée est toujours en relation avec l'état source,

— une étape se produit dans le programme cible et le programme source produit une erreur.

On rappelle que les définitions annexes sont disponibles dans la section 2.4 et que les termes complets (sans omission de paramètre _) sont accessibles dans le code Coq.

```
Lemma backward_sim:
  (∀ s t l t', stepr _ t l t' → full_rel s t →
    (∃ s', taustep _ s l s' ∧  full_rel s' t') ∨
    (tau _ l ∧  full_rel s t' ∧  full_order t' t) ∨
    (∃ s', tauscstar _ s s' ∧  can_do_error _ s')).
```

Ce lemme peut se traduire plus simplement à l'aide du diagramme suivant par une disjonction de trois cas :



En pratique, les propriétés que l'on vient de définir sont adaptées aux différentes preuve de compilation même si le cœur de la propriété est toujours présent. En pratique, on a deux preuves de la propriété de simulation arrière : l'une que l'on appelle à faible interaction mémoire et l'autre à forte interaction mémoire. Les deux sont nécessaires pour des passes de compilation différentes. En pratique, la simulation arrière à forte interaction mémoire est nécessaire pour une passe de compilation qui réorganise complètement l'allocation de la mémoire.

5.1.1 Simulation arrière à faible interaction mémoire

On se place dans un cas de simulation arrière à un pas pour lequel le module de simulation arrière à un pas par processus léger est simple. Dans la définition suivante, on omet certains types mais on respecte une convention de nommage. On appelle les états sources s, les états cibles t, les étiquettes l, la relation entre états r et l'ordre c. Cette propriété backward_sim_t combine un ordre bien-fondé c, une relation de blocage et une simulation arrière par étape. Cette relation de simulation arrière à un pas par processus léger comporte plus de cas que la

relation que l'on a présenté dans le début de ce chapitre. En effet, on a besoin de gérer des événements différemment pour certaines passes de compilation et on ajoute les cas pseudotau et fencetau.

Dans le cas pseudotau, un événement τ dans le langage cible peut représenter un événement de lecture d'une valeur dans le source. Dans certains cas, la compilation nous permet d'éviter des lectures dans le langage cible car on dispose déjà de la valeur en mémoire alors que ces lectures sont nécessaires dans le langage source. La traduction en Coq (qui suit) peut sembler inutilement verbeuse. Elle l'est pour laisser le contrôle de la valeur lue à la machine mémoire (ce qui implique de pouvoir lire n'importe quelle valeur dans la machine par processus léger). En Coq, on traduit cela en fixant une étiquette l0 faisant référence à un emplacement mémoire et une valeur donnée, mais on n'utilise pas cette valeur : on se contente de s'assurer que l'étiquette que l'on utilise l' est relié à l0 par samekind (et fait donc référence au même emplacement. Le cas fencetau traduit une fence qui a été ajoutée au cours de la compilation et qui se traduit par un événement tau dans le langage source.

```
Definition backward_sim_t (src tgt : lts lbl) (r : St tgt → St src → Prop)
  (c : St tgt → St tgt → Prop): Prop :=
   well_founded c ∧  (* ordre *)
   (∀ s t, stuck_t _ t → r t s → ev_stuck_or_error_t _ s) ∧  (* relation stuck *)
 (* relation backward_step *)
 (∀ s t l t', stepr _ t l t' →  r t s →
  (∃ s', taustep_t _ s l s' ∧  r t' s') ∨  (* etape avec source *)
  (tau lbl l ∧  r t' s ∧  c t' t) ∨  (* stuttering  *)
  (tau lbl l ∧  (∃ l0, ∀ l', samekind lbl l' l0 →
   ∃ s', taustep_t src s l' s' ∧  pseudotau lbl l' ∧  r t' s')) ∨  (* etape lecture *)
  (fencetau lbl l ∧  (∃ s', ∃ l', tau lbl l' ∧  taustep_t src s l' s' ∧  r t' s')) ∨  (* etape fence *)
  ev_stuck_or_error_t src s). (* erreur *)
```

Le jeu d'hypothèses du module Global_par_simple traduit la correction de la compilation de par processus légers parallèles (c'est à dire une propriété qui permet d'obtenir la conservation de traces une fois combinée aux propriétés concernant l'ordonnanceur et la machine mémoire). Elle contient une relation sur les états st_rel, une hypothèse sur les états initiaux thread_init_related, et une propriété de simulation arrière à un pas et blocage (global_backward_sim qui réutilise backward_sim_t ci-dessus). On omet les relation sur les environnements. La relation backward_sim_t est en fait paramétrée par un système de transition donné. mklts permet de créer le système de transition. Ici

mklts global_labels (Src.(SEM_G_step) lv correspond au système de transitions étiquetées du processus léger lv pour la composition parallèle de processus légers.

```
Module Type Global_par_simple (Src Tgt: SEM_G).

 Parameter st_rel    : Tgt.(SEM_G_ST) → Src.(SEM_G_ST) → Prop.
 Parameter st_order  : Tgt.(SEM_G_ST) → Tgt.(SEM_G_ST) → Prop.
 Parameter match_prg : Src.(SEM_G_PRG) → Tgt.(SEM_G_PRG) → Prop.

 Parameter thread_init_related:
  ∀ tgt_init fnp args tid
    (INIT : Tgt.(SEM_G_init) tid fnp args = Some tgt_init),
    ∃ src_init,
      Src.(SEM_G_init) tid fnp args = Some src_init ∧
      st_rel tgt_init src_init.

 Parameter global_backward_sim:
  ∀ lv,
    backward_sim_t
              (mklts global_labels (Src.(SEM_G_step) lv))
              (mklts global_labels (Tgt.(SEM_G_step) lv))
              st_rel
              st_order.

End Global_par_simple.
```

Ces propriétés sont grandement similaires à la propriété de simulation arrière à un pas que l'on définit sur les états de la machine globale. On définit la relation entre les états globaux sources et cibles de la façon suivante :

```
Definition full_rel sst tst  :=
    let (smem, srct, ssched) := sst in
    let (tmem, tgtt, tsched) := tst in
      mem_rel smem tmem ∧  sched_rel ssched tsched ∧  st_rel tgtt srct.
```

Cette relation est la plus simple possible : on ne fait qu'adjoindre les relations de simulation connues entre les différentes parties du langage. En effet, les relations de simulation des différents éléments sont nécessaires afin de pouvoir utiliser leurs hypothèses de simulation (simulation arrière à un pas). Cette relation de simulation est également suffisante, dans ce cas, pour montrer les propriétés sur la compilation par rapport à l'exécution de la machine globale.

On conserve également l'ordre donné sur les états cibles par la sémantique par processus légers :

```
Definition full_order t1 t2 :=
    let (ttso1, ttgt1, tsched1) := t1 in
    let (ttso2, ttgt2, tsched2) := t2 in
     st_order ttgt1 ttgt2
```

Le fait que cet ordre soit bien fondé est une conséquence triviale du fait que st_order le soit.

Preuve d'initialisation

L'équivalence des états d'initialisation se montre facilement en combinant les lemmes d'initialisation des différentes machines. L'état initial cible est le tuple des états initiaux des différentes machines. On utilise les hypothèses d'initialisation des différentes machines pour prouver l'existence d'états sources correspondants et on obtient facilement la correction de l'initialisation init_related.

Preuve de blocage

Pour prouver que le passage aux machines globales conserve cette propriété, on a besoin de définir précisément dans quel cas une machine globale peut être bloquée. On a donc besoin de pouvoir définir la notion d'accessibilité pour l'ordonnanceur : en l'absence de primitives de synchronisation lancée par le programme (lock, wait etc), on a besoin de savoir quels processus légers peuvent s'exécuter. On peut retrouver les définitions de schedstar à la section 2.4.

Définition 16 (Access) *Un processus léger tid est accessible depuis un état sched si, en l'absence de nouveaux appels à des primitives de synchronisation (par exemple, wait qui stoppe l'exécution d'un processus léger), il pourra s'exécuter après un nombre fini d'étapes d'autres processus léger.*

```
Definition access tid tsched :=
 ∃ tsched', ∃ tsched'',
   schedstar scheduler_step tsched tsched' ∧
   scheduler_step tsched' (SCstep tid) tsched''.
```

```
Definition te_sched_corres tid ev sev :=
match ev, sev with
| THsyscall id _ ltid lv, SCsys id1 tid1 ltid1 lv1 ⇒ id = id1 ∧  tid = tid1
| THsyscallfail id _ ltid lv, SCsys id1 tid1 ltid1 lv1 ⇒ id = id1 ∧  tid = tid1
```

```
| _, _ ⇒ False
end.
```

L'hypothèse de bloquage dans une machine globale devient :

```
Definition blocage sg : Prop :=
let (mem, tthr, tsched) := sg in
∀ tid, access _ tid tsched → (∀ l tthr',
    (∼ te_sys l ∨  ∼ step tid tthr l tthr') ∨
    (te_sys l ∧  step tid tthr l tthr' ∧
    ∀ tsched' tsched'' l', schedstarstep _ tsched tsched' →
     ∼ Tgt_EXT.scheduler_step tsched' l' tsched'' ∨  ∼ te_sched_corres tid l l'))
```

C'est-à-dire que pour qu'un état soit bloquant, il faut que tout processus léger accessible depuis cet état (par l'ordonnanceur) soit bloquant ou que l'étape faite par ce processus léger émette un événement système qui n'est pas rattrapé par un état accessible de l'ordonnanceur.

On démontre que cette hypothèse se valide en dépilant les actions faites par tout les processus légers. On procède par induction sur l'ensemble des processus légers grâce au prédicat active_thread de la machine ordonnanceur. Le reste est une preuve par cas qui nous permet d'établir que la relation full_rel combinée au blocage de l'état de la machine cible implique la propriété blocage énoncée ci-dessus. Cette propriété implique que l'état source est aussi bloquant et on clôt ainsi la preuve.

Propriété de simulation arrière à un pas

La preuve de simulation arrière à un pas est essentiellement une preuve par cas. Les abstractions des différentes machines ont en effet été construites autour d'une hypothèse principale de simulation arrière. Le schéma globale consiste pour une étape donnée à séparer l'état de la machine globale en état de la machine mémoire, de la sémantique par processus légers et de la machine ordonnanceur(la figure 5.1 rappelle les relations entre ces différentes parties).

On applique ensuite les hypothèses de type simulation arrière à un pas respective à ces parties d'états. Puis on regroupe les états source créés ou on génère les états d'erreurs/blocage au niveau de la machine source. Les propriétés utilisés pour la simulation arrière à un pas sont respectivement :

— pour le scheduler :

```
Parameter valid_sched_bs: ∀ tsched tsched' ssched ev,
  TgtE.scheduler_step tsched ev tsched' → sched_rel ssched tsched →
```

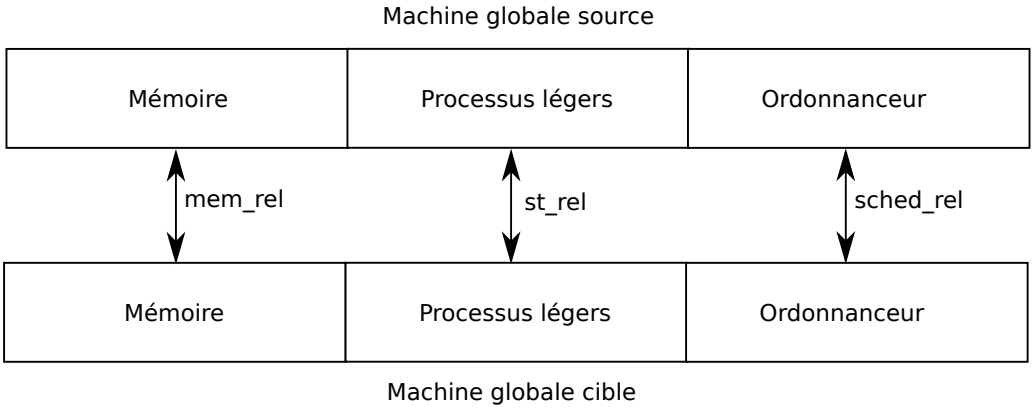


FIGURE 5.1 – *Relations de simulation entre machine source et cible*

```
( ∃ ssched'', ∃ ssched', @schedstar _ SrcE.scheduler_step ssched ssched' ∧
  ( (SrcE.scheduler_step ssched' ev ssched'' ∧  sched_rel ssched'' tsched') ∨
    (ev <> SCtau ∧  SrcE.scheduler_step ssched' SCfail ssched'') ) ).
```

— pour la mémoire :

```
Parameter valid_tso_bs: ∀ ttso ttso' stso ev,
 Ttgt.mem_step ttso ev ttso' → mem_rel stso ttso →
 ((∃ ev', mldo ev' ev ∧
     ∃ stso', tsotstep Tsrc.mem_step stso ev' stso' ∧  mem_rel stso' ttso')
 ∨  ∀ ev', mldi ev' ev →
   ∃ stso', tsotstep Tsrc.mem_step stso ev' stso' ∧  mem_rel stso' ttso').
```

— pour la machine par processus léger :

```
Parameter global_backward_sim:
 ∀ sge tge lv
  (GENR : genv_rel sge tge),
  backward_sim_t
            (mklts global_labels (Src.(SEM_G_step) sge lv))
            (mklts global_labels (Tgt.(SEM_G_step) tge lv))
            (st_rel sge tge)
            (st_order sge tge).
```

Cette preuve Coq contient de nombreux cas (combinaison des différentes étiquettes possibles) et certains d'entre eux nécessitent l'utilisation des hypothèses discutées dans le chapitre 4.

5.1.2 Simulation arrière à forte interaction mémoire

Dans le deuxième cas de simulation arrière à un pas, on a besoin d'hypothèses plus importantes sur les relations entre les états cibles et sources notam-

ment en mémoire. La relation de simulation arrière de la sémantique parallèle par processus est beaucoup moins intuitive car l'étape de compilation correspondante traduit un changement de l'allocation de la mémoire. On distingue en effet une disjonction de cas assez importante en fonction de l'événement lancé (voir le code Coq). Par exemple, pour une opération d'écriture, on utilise la propriété de write_simulationg :

Définition 17 (Présence dans la pile) *stack_pre est une définition annexe qui permet de différencier les événements mémoire influençant la pile des autres événements mémoires. Ces événements sont traçables grâce à deux listes d'événements mémoires: l'une faite par le programme source, l'autre par le programme cible. Ce prédicat stack_pre est vrai pour les emplacements mémoires qui sont sur la pile dans le programme cible mais qui ne le sont pas dans le source. Le prédicat chunk_inside_range_list traduit la présence du range obtenu à partir de (p, c) dans tp. range_not_in traduit le fait que ce range est disjoint de tout les autres dans sp.*

```
Definition stack_pre p c sp tp :=
 chunk_inside_range_list p c tp ∧  range_not_in (range_of_chunk p c) sp.
```

Définition 18 (Simulation pour l'écriture) *On donne write_simulationg qui est une partie (partie écriture en mémoire) de la simulation entre les états par processus légers complète. Pour une étape d'écriture dans la machine cible donné, on a soit :*

- *blocage ou erreur de la machine source,*
- *un événement τ si l'événement mémoire est stack_pre dans le source,*
- *une étape mémoire reliée dans le source.*

Avec ss l'état source, ts l'état cible, ts' l'état cible après l'étape d'écriture, tm une mémoire séquentielle, p et c représentant un emplacement mémoire et v une valeur, la propriété write_simulationg décrit le cas d'écriture pour la simulation arrière à un pas.

```
Definition write_simulationg tid ss ts ts' tm sp tp p c v :=
 stuck_or_error _ ss ∨
  (∃ tm',
   store_ptr c (tm tid) p v = Some tm' ∧
   (stack_pre p c (sp tid) (tp tid) ∧  ((∃ ss', src_taustep tid ss TEtau ss' ∧
             rel ts' tp (tupdatet tid tm' tm) ss' sp) ∨
    (rel ts' tp (tupdatet tid tm' tm) ss sp ∧  ord ts' ts)))) ∨
 (∃ v', Val.lessdef v' v ∧
          ∃ ss',
          src_taustep tid ss (TEmem (MEwrite p c v')) ss' ∧
```

```
        rel ts' tp tm ss' sp).
```

En combinant write_simulationg dans un des cas de local_intro_simulationg, on obtient la relation de simulation arrière à un pas par processus léger pour une étiquette write.

```
Definition local_intro_simulationg :=
 ∀ tid ss ts ts' tm tm' sp tp l,
  tgt_step tid ts l ts' →
  rel ts tp tm ss sp →
  match l with
   | TEtau              ⇒ tau_simulationg tid ss ts ts' tm sp tp
   | TEmem (MEwrite p c v) ⇒
     write_simulationg tid ss ts ts' tm sp tp p c v
  (* ... *)
  end.
```

On passe le reste des propriétés définissant local_intro_simulationg. Il y a des relations de simulation arrière différentes et aussi verbeuses que celles présentées ici. Ces relations sont présentes dans le code Coq et les exposer ici ne facilitent pas la compréhension. Par ailleurs, la propriété de simulation arrière (utilisé dans une seule passe de compilation) issue des prédicats par processus légers local_intro_simulationg reste à finaliser. On ne détaillera donc pas cette section plus avant.

5.2 Conservation de trace

Dans cette section, on va s'attacher à montrer que la simulation arrière à un pas parallèle de la section précédente implique la préservation de trace entre la machine globale cible et la machine globale source. On va tout d'abord donner la définition d'une trace d'exécution pour le compilateur certifié, puis on explique comment on peut passer de la propriété de simulation arrière à un pas à la conservation de trace. La préservation de trace définit la correction du compilateur et cette preuve clôt donc la preuve de compilation.

Définition 19 (Trace) *Une trace est définie comme une suite d'événements observables (générés par la sémantique par processus légers) potentiellement infinie. Des marqueurs sont utilisés pour reconnaître différents types de traces particulière : Send pour la terminaison du programme, Sinftau pour une boucle infinie du programme, Sinfsched pour une boucle infinie de l'ordonnanceur.*

```
CoInductive trace: Type :=
| Send
| Sinftau
| Sinfsched
| Scons (x: A) (t: trace)
```

Types de traces On crée des prédicats permettant de reconnaître différents types de trace pour les différencier.

Définition 20 *Une trace d'exécution peut-être une suite infinie d'événements silencieux (séparés par un nombre fini d'événements silencieux) :*

```
CoInductive inftau s: Prop :=
 | inftau_cons: ∀ s' l, tau l →
    taustep _ s l s' →
    inftau s' →
    inftau s.
```

Définition 21 *Une trace d'exécution peut être une suite infinie d'événements visibles (séparés par un nombre fini d'événements silencieux) :*

```
CoInductive infnontau s : trace event → Prop :=
| infnontau_cons: ∀ s' ev t,
   taustep _ s (GMvisible ev) s' →
   (ev <> Efail) →
   infnontau s' t →
   infnontau s (Scons ev t).
```

Définition 22 *Une trace d'exécution peut être une suite infinie d'événements d'ordonnancement silencieux séparé d'un nombre fini d'événements silencieux :*

```
CoInductive infsystau s: Prop :=
 | infsched_cons: ∀ s' l, tau_sys _ l →
    (∃ s'', tautsostar _ s s'' ∧  stepr _ s'' l s') →
    infsystau s' →
    infsystau s.
```

Définition 23 *Une trace d'exécution peut être une suite finie d'événements finissant sans erreur :*

```
Inductive fintrace s (l : list event) s' : Prop :=
| fintrace_refl:
    taustar _ s s' → l = nil → fintrace s l s'
| fintrace_trans: ∀ ev s'' l'
    (TS: taustep lts s (GMvisible ev) s'')
    (EV: ev <> Efail)
    (FIN: fintrace s'' l' s')
    (EQ: l = ev :: l'),
    fintrace s l s'.
```

On peut donc définir un prédicat trace qui représente exhaustivement les différentes traces d'exécution possibles. trace s t définit une trace d'exécution partant de s et faisant les événements de la liste (potentiellement infinie) t. Dans le cas d'une trace finie, le dernier élément de t traduit le comportement du programme (terminaison, boucle infinie, etc). Le cas d'erreur est particulier car toutes les traces commençant par une suite d'événements fini puis par un événements d'erreur sont considérées comme valide.

```
Inductive traces s t : Prop :=
| traces_end: ∀ l s'
    (PREF: fintrace s l s')
    (STUCK: ∀ l' s'', ∼ step s' l s'')
    (TEQ: l ++ Send = t),
    traces s t
| traces_fail: ∀ l s' s'' t'
    (PREF: fintrace s l s')
    (FAIL: step s' GMfail s'')
    (TEQ: l ++ t' = t),
    traces s t
| traces_inftau: ∀ l s'
    (PREF: fintrace s l s')
    (INF: inftau s')
    (TEQ: l ++ Sinftau = t),
    traces s t
| traces_infschedtau: ∀ l s'
    (PREF: fintrace s l s')
    (INF: infschedtau s')
    (TEQ: l ++ Sinfsched = t),
    traces s t
| traces_infnontau: ∀
    (INF: infnontau s t),
```

```
  traces s t.
```

Pour un programme on définit le prog_traces p args t qui prend un programme des arguments et une trace et qui est valide si la trace est celle du programme.

```
Definition prog_traces p args t :=
 ∃ s, init _ p args s ∧
  traces _ s t.
```

Avec match_prg la propriété validant que tprg est le programme compilé de sprg et prog_traces la définition de trace définit ci-dessus, on définit la notion d'inclusion de trace de la manière suivante :

```
Theorem traces_incl:
 ∀ {sprg tprg args t}
  (MP: match_prg sprg tprg)
  (T: prog_traces Tgt tprg args t),
  prog_traces Src sprg args t.
```

C'est à dire que si un programme compilé a une trace t alors le programme source a également cette trace t ou alors le programme source comporte une erreur (cas traces_fail dans traces qui, après une erreur du programme, permet de concaténer n'importe quelle trace).

La démonstration de l'inclusion de traces (traces_incl) se fait par une disjonction de cas sur la propriété traces et par l'utilisation dans chaque cas des propriétés de simulation arrière à un pas démontrées dans la section précédente de ce chapitre.

Conclusion

Dans ce chapitre, on a montré que la préservation de traces est possible pour les langages de notre formalisme, c'est-à-dire, la combinaison dans une machine globale d'un machine mémoire, d'un sémantique par processus légers, et d'un machine ordonnanceur.

Pour cela, on réutilise les démonstrations de correction de compilation séquentielle (de Compcert) que l'on adapte et que l'on combine avec les hypothèses sur les abstractions de la machine mémoire et de l'ordonnanceur pour obtenir une simulation arrière pour les machines globales. On montre ensuite que la préservation de traces d'exécution découle de la notion de simulation arrière pour les machines globales.

6 Instantiations des modèles

Sommaire

Ce chapitre permet de garantir que les abstractions que l'on définit dans le chapitre 4 peuvent être utilisées avec des modèles mémoires et des ordonnanceurs abstraits représentant le comportement réel d'ordinateurs. On va détailler plusieurs modèles mémoires qui nous semblent pertinents: un modèle séquentiellement consistant 6.1.1 et un modèle de type TSO (on rappelle que ce travail est basé sur CompcertTSO). Pour l'ordonnanceur, on montre qu'un ordonnancement abstrait permettant l'utilisation de primitives de synchronisation basique est possible : on se limite aux primitives wait/notify.

6.1 Modèles mémoires

Les modèles mémoire que l'on présente sont complémentaires. Le modèle TSO est un modèle très proche de la machine qui représente finement le comportement d'un processeur réel. Mais, pour un développeur qui voudrait se baser sur la spécification Coq pour s'assurer de la compilation correcte de ses programmes, ce modèle peut être difficile à utiliser. Le développeur devra en permanence vérifier que des unbuffering ne contrarient pas le parallélisme de son programme (il pourrait se baser sur des hypothèses plus larges de compatibilité

avec TSO mais ces hypothèses ne sont pas formalisées en Coq). Il serait plus simple d'utiliser un modèle séquentiellement consistant pour le langage source et, au cours de la compilation, de le traduire vers un langage cible avec un modèle TSO. Cela permet une relative simplicité du langage pour le développeur tout en garantissant la compatibilité avec la machine cible. Les deux modèles sont présentés comme des instances des spécifications de modèles mémoire que l'on a définies dans la section 4.4. Ils sont fortement similaires, seule l'action d'unbuffering changeant d'un modèle à l'autre, on omettra donc certaines redondances.

6.1.1 Modèle séquentiellement consistant

Définition d'une machine séquentiellement consistante Dans ce modèle, on va chercher à forcer une forme de consistance séquentielle : tout se passe comme si les effets en mémoire étaient exécutés dans un ordre séquentiel consistant avec le programme. L'idée est donc d'utiliser une mémoire partagée à accès direct pour forcer cette propriété. On retrouve la sémantique classique d'entrelacement dans le contexte de notre modèle mémoire abstrait. On ajoute également dans l'état du modèle mémoire une liste d'identifiants de processus légers déjà utilisées (les démonstrations sont plus simples lors de la création de processus léger par exemple) :

```
Definition mem_state:= (mem * list thread_id)%type.
```

On fournit un prédicat fournissant un aperçu de la mémoire vu par un processus léger comme défini dans l'abstraction. Dans notre cas, pour le prédicat per_proc_memory, tous les processus légers renvoient la même mémoire globale.

```
Definition per_proc_memory (m: mem_state) (tid: thread_id) := fst m.
```

On définit thread_alive comme la seconde partie de l'état :

```
Definition tso_alive (m: mem_state) := snd m.
```

On définit les actions en mémoire classique (write, alloc, free) comme les actions mémoires séquentielles auxquelles elles correspondent. On ajoute également certains cas d'erreur qui correspondent à des cas d'erreur déjà présent dans le modèle séquentiel (défaut d'allocation lors d'une écriture etc). Pour simplifier, à chaque action mémoire faite par un processus léger, on ajoute ce processus léger à l'ensemble des processus légers vivants s'il n'y est pas. On notera que l'on ne met pas d'action read−modify−write car, les accès mémoire sont directs et elle n'est donc pas utile. Le début de la définition de l'inductif mem_step' est donc :

```
Inductive mem_step': mem_state → MMevent → mem_state → Prop :=
| write: ∀ p c v tid m m',
  snd m' = insert_l (snd m) tid →
  store_ptr c (fst m) p v = Some (fst m') →
  mem_step' m (MMmem tid (MEwrite p c v)) m'
| read: ∀ p c v tid m,
  load_ptr c (fst m) p = Some v →
  mem_step' m (MMmem tid (MEread p c v)) m
| alloc: ∀ p n k tid m m',
  snd m' = insert_l (snd m) tid →
  alloc_ptr (p, n) k (fst m) = Some (fst m') →
  mem_step' m (MMmem tid (MEalloc p n k)) m'
| free: ∀ p k tid m m',
  snd m' = insert_l (snd m) tid →
  free_ptr p k (fst m) = Some (fst m') →
  mem_step' m (MMmem tid (MEfree p k)) m'
| write_fail: ∀ p c v tid m,
  store_ptr c (fst m) p v = None →
  mem_step' m (MMmemfail tid (MEwrite p c v)) m
| read_fail: ∀ p c v tid m, load_ptr c (fst m) p = None →
  mem_step' m (MMmemfail tid (MEread p c v)) m
| alloc_oom: ∀ p n k tid m, alloc_ptr (p, n) k (fst m) = None →
  mem_step' m (MMoutofmem tid n k) m
| free_fail: ∀ p k tid m, free_ptr p k (fst m) = None →
  mem_step' m (MMmemfail tid (MEfree p k)) m
```

Les derniers constructeurs de cet inductif sont args_fail et mem_sys. Ils font référence aux événements systèmes et nécessitent l'introduction de prédicats permettant de vérifier et renvoyer la valeur des arguments demandés par un appel donné. Pour cela, on réutilise les inductifs tso_arglist et tso_arglist_fail (de CompcertTSO) qui permettent de contrôler la valeur d'arguments. Ces inductifs prennent un état mémoire, une liste d'arguments appelés, et une liste de valeurs retournées et vérifient que la liste des valeurs retournées correspond bien aux lectures faites par la liste d'arguments. tso_arglist_fail correspond au cas d'échec de ce processus. Le constructeur tso_arglist_val correspond à une lecture directe (cas d'un language haut-niveau ne passant pas les arguments en mémoire) et tso_arglist_read correspond à une lecture en mémoire (cas de language bas-niveau).

```
Inductive tso_arglist: mem→list(pointer∗memory_chunk+val)→list val→mem→Prop:=
```

```
| tso_arglist_refl: ∀ tso,  tso_arglist tso nil nil tso
| tso_arglist_val:  ∀ tso locs vals tso' v (MA: tso_arglist tso locs vals tso'),
   tso_arglist tso (inr (pointer * memory_chunk) v :: locs) (v :: vals) tso'
| tso_arglist_read: ∀ tso p c v locs vals tso'' (RD :  load_ptr c tso p = Some v)
   (MA: tso_arglist tso locs vals tso''),
   tso_arglist tso (inl val (p, c) :: locs) (v :: vals) tso''.

Inductive tso_arglist_fail : mem → list (pointer * memory_chunk + val) → Prop :=
| tso_arglist_fail_err: ∀ tso p c rest (TST: load_ptr c tso p = None),
   tso_arglist_fail tso (inl val (p, c) :: rest)
| tso_arglist_fail_val:  ∀ tso locs v (MA: tso_arglist_fail tso locs),
   tso_arglist_fail tso (inr (pointer * memory_chunk) v :: locs)
| tso_arglist_fail_read: ∀ tso p c v locs (RD : load_ptr c tso p = Some v)
   (MA: tso_arglist_fail tso locs),
   tso_arglist_fail tso (inl val (p, c) :: locs).
```

On ajoute donc deux constructeurs correspondant respectivement à la réussite ou l'échec de la lecture des arguments:

```
| args_fail: ∀ tid id realargs args ltid m,
 tso_arglist_fail (fst m) realargs →
   mem_step' m (MMsysfail tid id realargs args ltid) m

| mem_sys: ∀ tid id realargs args ltid m m',
 tso_arglist (fst m) realargs args (fst m') →
 snd m = snd m' →
   mem_step' m (MMsys tid id realargs args ltid) m'
```

On définit l'initialisation mem_init comme le couple d'une mémoire donnée et d'un processus léger unique débutant l'exécution :

```
Definition mem_init (m: mem) (ms: mem_state) := ∃ tid, ms = (m, tid::nil).
```

Les hypothèses formulées dans l'abstraction ne posent pas de problèmes dans ce modèle. Leurs difficultés sont liées à l'utilisation d'un modèle mémoire faible ce qui explique leur simplicité dans notre cas. Par exemple, les hypothèses de coherence_oth et coherence_seq qui imposent qu'après un événement mémoire, la mémoire de chaque processus léger change en conséquence ou reste inchangée, deviennent triviales. On a également bien choisi la définition de la machine pour prouver facilement les hypothèses concernant never_stuck. Donc le reste de la spécification de la machine ne pose pas de difficultés.

Instanciation de la relation de simulation entre machine mémoire Dans cette partie, on va montrer que si on a deux modèles mémoire séquentiellement consistants, on peut bien les utiliser pour démontrer que la passe de compilation correspondante est correcte. En effet, il est possible qu'on instancie de façon correcte un modèle mémoire mais qu'il soit impossible d'instancier le module de relation à cause de contraintes trop fortes dans ce module. On va donc vérifier que l'on peut créer le module (rel_seq_mem_inst) le plus simple possible en prenant deux machines mémoire séquentiellement consistantes comme source et cible.

```
Module rel_seq_mem_inst <: mem_rel_1 Seqmem Seqmem Mio_sig_eq_inst.
```

Une relation entre états source et cible simple est l'égalité :

```
Definition mem_rel (s t: Seqmem.mem_state) := s = t.
```

Avec l'égalité comme relation, les relations de backward_simulation sont extrêmement simple car il suffit d'effectuer les mêmes étapes dans l'état source et l'état cible pour que ces états restent reliés au cours de l'exécution. Cela conclut donc la viabilité du modèle mémoire que l'on vient de définir dans le contexte de l'extension de compilateur que l'on présente dans ce manuscrit.

6.1.2 Modèle TSO

Instantiation de la machine mémoire Le modèle abstrait du chapitre 4 étant inspiré du modèle TSO, il est naturel de vérifier que ce dernier est bien une instance de notre abstraction. On va donc reprendre les définitions du modèle TSO car il fournit un modèle mémoire faible fidèle au comportement de certains processeurs multi-cœurs :

On définit les états comme la combinaison d'une mémoire globale et d'un buffer d'actions en mémoire pour chaque processus léger:

```
Record tso_state' := mktsostate
 { tso_buffers : buffers
 ; tso_mem : mem
 }.
```

avec les buffers des listes de buffer_item contenan les 3 types d'action en mémoire élémentaire :

```
Inductive buffer_item : Type :=
 | BufferedWrite (p: pointer) (c: memory_chunk) (v: val)
 | BufferedAlloc (p: pointer) (i: int) (k: mobject_kind)
```

```
| BufferedFree  (p: pointer) (k: mobject_kind).

Definition buffers := thread_id → list buffer_item.
```

On complète cette définition initiale (présente dans l'implémentation CompcertTSO) avec la liste d'identifiants de processus léger actifs : on utilise un nouvel identifiant de processus léger pour chaque création car on a besoin de connaître les processus légers qui ont déjà été créés. Cette liste ne doit pas contenir de doublons et être compatible avec tso_state' : un identifiant correspondant à un buffer non vide doit appartenir à la liste. En coq, avec l cette liste et tso de type tso_state', la propriété s'écrit :

```
NoDup l ∧  (∀ tid, tso_buffers tso tid <> nil → In tid l)
```

On reprend également le prédicat d'unbuffer_safe de TSO sur les éléments de type tso_state' qui garantit que toute combinaison d'unbuffer pour un état donné ne mène pas à une erreur. C'est un prédicat de bonne formation de l'état qui assure que les unbuffering ne causeront pas une erreur. C'est à dire que cette propriété assure, pour un tso_state', qu'il est possible de faire des unbuffer successivement sur n'importe quel processus léger sans causer d'erreurs. Par exemple, cette propriété interdit qu'un état contiennent une allocation identique dans les buffers de deux processus légers différents. Si c'était le cas, l'unbuffering (le passage en mémoire globale) de ces deux allocation successivement génère une erreur à la deuxième allocation car on ne peut pas allouer deux fois un emplacement sans l'avoir libéré. La propriété d'unbuffer_safe ne serait pas valide dans le cas de la succession directe de deux événements d' unbuffer sur ces allocations. Il découle de cette propriété sur les états que la gestion des erreurs se fait directement : on n'attend jamais un unbuffering avant de lancer une erreur.

```
Inductive unbuffer_safe : tso_state' → Prop :=
| unbuffer_safe_unbuffer:
 ∀ tso
   (ABIS: ∀ t bi b, tso.(tso_buffers) t = bi :: b →
     ∃ m', apply_buffer_item bi tso.(tso_mem) = Some m')
   (UNBS: ∀ t bi b m',
     tso.(tso_buffers) t = bi :: b →
     apply_buffer_item bi tso.(tso_mem) = Some m' →
     unbuffer_safe (mktsostate (tupdate t b tso.(tso_buffers)) m')),
   unbuffer_safe tso.
```

Cela nous permet donc de définir un état pour la machine mémoire.

```
Definition mem_state := {t : (tso_state' ∗ list thread_id) |
```

```
unbuffer_safe (fst t) ∧ NoDup (snd t) ∧ (∀ tid, tso_buffers (fst t) tid <> nil → In tid (snd t))}.
```

On définit le prédicat d'initialisation comme valide si l'argument donné est une mémoire globale vide avec des buffers vide (avec unbuffer_safe_nil la propriété de correction d'un état avec mémoire et liste vide) :

```
Definition tso_init (m : mem) : mem_state :=
 exist _ ((mktsostate (fun t ⇒ nil) m), nil) (unbuffer_safe_nil m).
```

On définit per_proc_memory comme la mémoire obtenue par l'application du buffer local au processus léger à la mémoire globale. La fonction apply_buffer' exécute cette application

```
Definition apply_buffer' (b: list buffer_item) (m: mem) (H: ∃ m', apply_buffer b m = Some m') :=
match (apply_buffer b m) with
| Some m' ⇒ m'
| None ⇒ default
end.
```

La lemme unbuffer_apply permet de récupérer la propriété d'unbuffer_safe d'une mem_state ce qui nous permet de typer correctement l'appel de apply_buffer' pour l'utiliser dans per_proc_memory :

```
Definition per_proc_memory (tso: tso_state) tid :=
 apply_buffer' (tso_buffers (fproj1_sig tso) tid)
                (tso_mem (fproj1_sig tso)) (unbuffer_apply tso tid).
```

On définit tso_alive comme la liste d'identifiants de processus légers que l'on a adjoint à l'état de la machine mémoire.

```
Definition tso_alive (t: tso_state) := snd (proj1_sig t).
```

On va également définir le prédicat d'exécution d'une telle machine en s'inspirant largement de TSO. On va découper l'inductif de définition afin de pouvoir expliquer les étapes possibles en fonction de l'état de la machine.

```
Inductive tso_step' : tso_state → tso_event → tso_state → Prop
```

Opérations de base On définit les opérations mémoires classiques (write, alloc, free) comme l'ajout dans le buffer du processus léger s'exécutant une opération buffer correspondante (BufferedWrite, BufferedAlloc et BufferedFree) :

```
| tso_step_write : (* Memory write  (goes into buffer) *)
  ∀ t ts ts' cts' p c v
    (EQts': ts' = buffer_insert (fproj1_sig ts) t (BufferedWrite p c v))
    (SAFE: unbuffer_safe ts')
```

```
    (Ceq: fproj1_sig cts' = ts')
    (Leq: sproj1_sig cts' = insert_l (sproj1_sig ts) t),
  tso_step' ts (MMmem t (MEwrite p c v)) cts'

| tso_step_read :   (* Memory read *)
  ∀ ts tsi t m' p c v
    (EQ: tsi = fproj1_sig ts)
    (AB: apply_buffer (tsi.(tso_buffers) t) tsi.(tso_mem) = Some m')
    (LD: load_ptr c m' p = Some v),
  tso_step' ts (MMmem t (MEread p c v)) ts

| tso_step_alloc :  (* Memory allocation  (goes into buffer) *)
  ∀ t ts cts' ts' p i k
    (EQts': ts' = buffer_insert (fproj1_sig ts) t (BufferedAlloc p i k))
    (UNS: unbuffer_safe ts')
    (Ceq: fproj1_sig cts' = ts')
    (Leq: sproj1_sig cts' = insert_l (sproj1_sig ts) t),
  tso_step' ts (MMmem t (MEalloc p i k)) cts'

| tso_step_free :  (* Memory deallocation (goes into buffer) *)
  ∀ t ts cts' ts' p k
    (EQts': ts' = buffer_insert (fproj1_sig ts) t (BufferedFree p k))
    (UNS: unbuffer_safe ts')
    (Ceq: fproj1_sig cts' = ts')
    (Leq: sproj1_sig cts' = insert_l (sproj1_sig ts) t),
  tso_step' ts (MMmem t (MEfree p k)) cts'
```

Cas d'erreurs pour les opérations de base On définit les cas d'erreurs pour ces mêmes opérations. Dans TSO, le cas d'erreur implique forcément que le buffer associé soit vide. En pratique, on pourra se ramener à ce cas de buffer vide dans les preuves en forçant plusieurs événements MMtau successifs. Dans cette instance, on s'attache également à coller le plus possible au modèle défini par CompcertTSO afin de justifier que les preuves passent toujours :

Un cas d'erreur se produit si la lecture de la mémoire séquentielle ne peut pas se faire (non alloué etc) :

```
| tso_step_read_fail: (* Memory read failure *)
  ∀ ts tsi t p c v
    (EQ: tsi = fproj1_sig ts)
    (Bemp: tsi.(tso_buffers) t = nil)
```

```
    (LD: load_ptr c tsi.(tso_mem) p = None),
  tso_step' ts (MMmemfail t (MEread p c v)) ts
```

Le cas d'erreur d'écriture correspond à une allocation non correcte (renvoie None) :

```
| tso_step_write_fail:
  ∀ ts tsi t p c v
    (EQ: tsi = fproj1_sig ts)
    (Bemp: tsi.(tso_buffers) t = nil)
    (LD: store_ptr c tsi.(tso_mem) p v = None),
  tso_step' ts (MMmemfail t (MEwrite p c v)) ts
```

Un cas d'erreur de la primitive read−modify−write se produit si l'on ne peut pas lire l'emplacement mémoire :

```
| tso_step_rmw_fail:
  ∀ ts tsi t p c v instr
    (EQ: tsi = fproj1_sig ts)
    (Bemp: tsi.(tso_buffers) t = nil)
    (LD: load_ptr c tsi.(tso_mem) p = None),
  tso_step' ts (MMmemfail t (MErmw p c v instr)) ts
```

On considère qu'un free est une erreur s'il ne peut pas s'exécuter (renvoie None) ou s'il existe un write d'un autre processus léger vers cet emplacement :

```
| tso_step_free_fail : (* Memory deallocation fail *)
  ∀ t ts ts' p k
    (EQ: fproj1_sig ts = ts')
    (Bemp: ts'.(tso_buffers) t = nil)
    (FAIL: match free_ptr p k (tso_mem ts') with
        | None ⇒ True
        | Some m' ⇒ ∃ tid', ∃ p, ∃ c, ∃ v, ∃ b,
                      tso_buffers ts' tid' = BufferedWrite p c v :: b
                  ∧  store_ptr c m' p v = None end),
      tso_step' ts (MMmemfail t (MEfree p k)) ts
```

Si on ne peut plus allouer de nouveaux éléments, on n'a plus de mémoire disponible :

```
| tso_step_outofmem :
   ∀ t ts n k
    (OOM: ∀ p,
          ~ unbuffer_safe (buffer_insert (fproj1_sig ts) t (BufferedAlloc p n k))),
  tso_step' ts (MMoutofmem t n k) ts
```

Instructions atomiques Les instructions atomiques ont un rôle particulier car elles permettent des synchronisations bas-niveau dans TSO. Dans notre implémentation, on a aussi la possibilité de faire des synchronisations grâce aux appels aux fonctions externes. Ces instructions atomiques permettent des synchronisations à un niveau beaucoup plus bas : elles utilisent le modèle mémoire pour se synchroniser (elles n'utilisent pas l'ordonnanceur).

```
| tso_step_mfence : (* Mfence (l'instruction ne vide pas le buffer,
elle ne s'execute pas tant qu'il n'est pas vide) *)
 ∀ ts t (Bemp: (fproj1_sig ts).(tso_buffers) t = nil),
 tso_step' ts (TSOmem t MEfence) ts

| tso_step_rmw : (* Read−modify−write (l'instruction ne vide pas le buffer) *)
 ∀ tsr tsr' ts ts' t p c v instr m'
   (EQ: ts = fproj1_sig tsr)
   (EQ': ts' = fproj1_sig tsr')
   (Bemp: ts.(tso_buffers) t = nil)
   (LD: load_ptr c ts.(tso_mem) p = Some v)
   (STO: store_ptr c ts.(tso_mem) p (rmw_instr_semantics instr v) = Some m')
   (EQts': mktsostate ts.(tso_buffers) m' = ts')
   (Leq: sproj1_sig tsr' = insert_l (sproj1_sig tsr) t),
 tso_step' tsr (TSOmem t (MErmw p c v instr)) tsr'
```

Unbuffering La notion d'unbuffering permet de propager les actions mémoires dans un ordre arbitraire entre les processus légers. C'est à dire que chaque processus léger peut initier un unbuffering à tout moment.

```
| tso_step_unbuffer : (* Apply buffer item *)
  ∀ t (ts cts: mem_state) ts' bufs' bi b m'
    (EQ: ts' = fproj1_sig ts)
    (EQbufs: ts'.(tso_buffers) t = bi :: b)
    (EQbufs': bufs' = tupdate t b ts'.(tso_buffers))
    (AB: apply_buffer_item bi ts'.(tso_mem) = Some m')
    (HCeq: fproj1_sig cts = {| tso_buffers := bufs'; tso_mem := m'|})
    (Leq: sproj1_sig cts = insert_l (sproj1_sig ts) t),
  tso_step' ts MMtau cts
```

Appels systèmes On va réutiliser tso_arglist_fail et tso_arglist légèrement modifiée pour coller au modèle et on rappelle donc leurs définitions.

```
Inductive tso_arglist_fail: tso_state → thread_id →
                        list (pointer * memory_chunk + val) → Prop :=
| tso_arglist_fail_err: ∀ t ts tsi p c rest
   (EQ: tsi = fproj1_sig ts)
   (Bemp: tso_buffers tsi t = nil)
   (LD: load_ptr c (tso_mem tsi) p = None),
   tso_arglist_fail ts t (inl val (p, c) :: rest)
| tso_arglist_fail_val:  ∀ tso t locs v
   (MA: tso_arglist_fail tso t locs),
   tso_arglist_fail tso t (inr (pointer * memory_chunk) v :: locs)
| tso_arglist_fail_read: ∀ tso tsi t p c v locs
   (EQ: tsi = fproj1_sig tso)
   (ABex: ∃ m', apply_buffer (tso_buffers tsi t) (tso_mem tsi) = Some m' ∧
   load_ptr c m' p = Some v)
   (MA: tso_arglist_fail tso t locs),
   tso_arglist_fail tso t (inl val (p, c) :: locs).
```

```
Inductive tso_arglist : mem_state → thread_id →
                    list (pointer * memory_chunk + val) → list val →
                    mem_state → Prop :=
| tso_arglist_refl: ∀ t tso,
   tso_arglist tso t nil nil tso
| tso_arglist_val:  ∀ tso t locs vals tso' v
   (MA: tso_arglist tso t locs vals tso'),
   tso_arglist tso t (inr (pointer * memory_chunk) v :: locs) (v :: vals) tso'
| tso_arglist_read: ∀ tsi t p c v tso' locs vals tso''
   (EQ: tsi = fproj1_sig tso')
   (ABex: ∃ m', apply_buffer (tso_buffers tsi t) (tso_mem tsi) = Some m' ∧
   load_ptr c m' p = Some v)
   (MA: tso_arglist tso' t locs vals tso''),
   tso_arglist tso' t (inl val (p, c) :: locs) (v :: vals) tso''.
```

On a unifié les appels de fonctions (par rapport à TSO qui avait des appels différents pour les créations de processus légers et les appels systèmes). On a aussi unifié la façon de propager les arguments à ces fonctions grâce aux prédicats tso_sys et tso_sys_fail. On notera que cette fonction dépendra des fonctions systèmes que l'on veut considérer et de celles que l'on considérera comme des entrées/sorties. Ici et pour des raisons de simplicité, on ne traduit que l'appel système identifié par 1 comme un appel avec un comportement particulier. Il correspond à la création d'un processus léger. On utilise le module signature_inst

pour définir la signature attendue pour un tel appel (ie: on doit avoir un pointeur vers le corps de la fonction et le reste des valeurs de l'appel sont les arguments passés à la fonction). La primitive tso_arglist permet de faire correspondre la liste des éléments (list (pointer * memory_chunk + val)) à des valeurs réelles list val.

```
Inductive tso_sys : mem_state→thread_id→positive→list(pointer*memory_chunk+val)→
 list val → list thread_id → mem_state → Prop :=
(* id = 1: supposed to create all the new threads that are in ltid *)
| creation: ∀ tso tid locs vals ltid tso',
 tso_arglist tso tid locs vals tso' →
 ~ In Vundef (signature_inst.non_undef 1%positive vals) →
 (∀ tid, In tid ltid → ~ In tid (tso_alive tso)) →
 tso_sys tso tid 1%positive locs vals ltid (alive_add ltid tso')
| ext_call: ∀ tso tid locs vals ltid tso' id (Nud: ~In Vundef (signature_inst.non_undef id vals)),
 id <> 1%positive →
 tso_arglist tso tid locs vals tso' →
 tso_sys tso tid id locs vals ltid (alive_add ltid tso').
```

```
Definition tso_sys_fail tso tid (id: positive) locs (lv: list val) (ltid: list thread_id) :=
 tso_arglist_fail tso tid locs ∨
 (∃ tso', tso_arglist tso tid locs lv tso' ∧ In Vundef (signature_inst.non_undef id lv))
 ∨ ∃ tid, In tid ltid ∧ In tid (tso_alive tso).
```

La traduction dans l'inductif global est la suivante :

```
| tso_step_ext : ∀ ts ts' tid id lpmv lv lt,
 tso_sys ts tid id lpmv lv lt ts' →
 tso_step' ts (TSOsys tid id lpmv lv lt) ts'

| tso_step_ext_fail: ∀ ts ts' tid id lpmv lv lt,
 tso_sys_fail ts tid id lpmv lv lt →
 fproj1_sig ts' = fproj1_sig ts →
 sproj1_sig ts' = sproj1_sig (alive_add lt ts) →
 tso_step' ts (TSOsysfail tid id lpmv lv lt) ts'.
```

La définition de tso_step nous permet de vérifier assez facilement les différentes propriétés de cohérence sur per_proc_memory, tso_alive et tso_init.

Instance de la relation de simulation entre machine mémoire Dans cette partie, on va montrer que si l'on a deux machines TSO, on peut les réutiliser en utilisant l'une comme machine mémoire source et l'autre comme machine mémoire cible. Cela nous permettra de les réutiliser dans nos preuves de correction : sans

ce module, c'est impossible. Comme précédemment, pour le modèle mémoire séquentiellement consistant on va utiliser une relation la plus simple possible sur les modèles.

On va donc montrer que l'on peut écrire un module de relation entre machine mémoire source et cible (avec MMmem le machine mémoire et Mio_inst_eq une instance de Mem_io)

```
Module MMmemsim_1 <: tso_rel_1 Si MMmem MMmem Mio_inst_eq.
```

On définit la relation entre états la plus simple possible en choisissant l'égalité comme relation de simulation :

```
Definition tso_rel := @eq TSOmem.tso_state.
```

Les propriétés définies dans le chapitre 4 se démontrent très facilement : les preuves sont toujours directes et consistent à dire que les états étant identiques, on peut simuler des étapes identiques. Ce résultat suffit à démontrer que notre instance est utilisable. D'autres relations nécessaires aux étapes de compilation à fortes interactions mémoires sont omises.

6.2 Ordonnanceur

Le but de cette section est de définir une instance d'ordonnanceur pour l'abstraction d'ordonnanceur du chapitre 4. Pour cela, on commence par définir une abstraction forte qui contraint fortement l'ordonnancement, on démontre ensuite que cette abstraction instancie bien le modèle abstrait du chapitre 4 et enfin on l'instancie par un ordonnanceur concret.

6.2.1 Simplification de l'abstraction

Dans les cas simples, on peut utiliser une spécification plus forte sur la machine ordonnanceur que celle du chapitre 4 : les étapes étiquetées par un SCtau ou SCstep n'influencent pas l'état de la machine ordonnanceur. On se passe donc de prédicats comme access et on simplifie grandement l'écriture de l'instance. En Coq:

```
Hypothesis tau_eq_s: ∀ ssched ssched',
  SrcE.scheduler_step ssched SCtau ssched' → ssched = ssched'.
```

On pourrait être plus subtil en définissant une relation d'équivalence sur les états conservés par des étapes étiquetées par SCtau ou SCstep. Cela nous permettrait d'avoir des instances d'ordonnanceur non-déterministes.

Définition 24 (Abstraction faible de relation d'ordonnancement) *L'abstraction de relation d'ordonnancement faible se traduit de manière simple en conservant une relation sched_rel, une simulation arrière à un pas valid_sched_bs, une initialisation sched_rel_init et en ajoutant des hypothèses fortes de conservation de l'état après un SCtau ou un SCstep.*

```
Module Type Simpl_sem (SrcE TgtE: Ext).

(** Relation between the scheduler states of the source and target *)
Variable sched_rel: SrcE.sched → TgtE.sched → Prop.

(** "backward simulation" for the scheduler *)
Hypothesis valid_sched_bs: ∀ tsched tsched' ssched ev,
  TgtE.scheduler_step tsched ev tsched' → sched_rel ssched tsched →
  (∃ ssched', SrcE.scheduler_step ssched ev ssched' ∧  sched_rel ssched' tsched') ∨
  (ev <> SCtau ∧  ∃ ssched', ∃ ssched'',
      @schedstar _ SrcE.scheduler_step ssched ssched' ∧
      SrcE.scheduler_step ssched' SCfail ssched'').

(** Initialisation properties *)
Hypothesis sched_rel_init: ∀ tid tsched, TgtE.ext_init tid tsched →
  ∃ ssched, SrcE.ext_init tid ssched ∧  sched_rel ssched tsched.

Hypothesis decide_full_rel: ∀ ssched tsched sev,
  sched_rel ssched tsched →
    SrcE.decide_full ssched sev = TgtE.decide_full tsched sev.

Hypothesis sched_stuck: ∀ ssched ssched' tsched ev,
  sched_rel ssched tsched → SrcE.scheduler_step ssched ev ssched' →
    ∃ tsched', ∃ tsched'', @schedstar _ TgtE.scheduler_step tsched tsched' ∧
      TgtE.scheduler_step tsched' ev tsched''.

Hypothesis tau_eq_s: ∀ ssched ssched',
  SrcE.scheduler_step ssched SCtau ssched' → ssched = ssched'.

Hypothesis step_eq_s: ∀ ssched ssched' tid,
```

```
  SrcE.scheduler_step ssched (SCstep tid) ssched' → ssched = ssched'.

Hypothesis tau_eq_t: ∀ ssched ssched',
  TgtE.scheduler_step ssched SCtau ssched' → ssched = ssched'.

Hypothesis step_eq_t: ∀ ssched ssched' tid,
  TgtE.scheduler_step ssched (SCstep tid) ssched' → ssched = ssched'.

Hypothesis sys_step: ∀ sched sched' id tid ltid args args0
  (VD: Val.lessdef_list args0 args),
  SrcE.scheduler_step sched (SCsys id tid ltid args) sched' →
  SrcE.scheduler_step sched (SCsys id tid ltid args0) sched'.

Hypothesis sys_decide_full: ∀ sched id tid ltid args args0
  (VD: Val.lessdef_list args0 args),
  SrcE.decide_full sched (SCsys id tid ltid args) =
    SrcE.decide_full sched (SCsys id tid ltid args0) ∨
  SrcE.decide_full sched (SCsys id tid ltid args0) = Evisible Efail.

End Simpl_sem.
```

On a vérifié en Coq que cette abstraction est une instance de l'abstraction générale que l'on définit à la section 4.5. Les démonstrations sont très simples car les hypothèses tau_eq_s et step_eq_s simplifient toutes les exécutions et rendent triviale l'abstraction faible. L'abstraction 24 est encore une abstraction cohérente pour un ordonnanceur réel que l'on verra dans la section suivante 6.2.3.

6.2.2 Modèle d'ordonnanceur

On définit un ordonnanceur le plus simple possible et on n'implémente que deux primitives de synchronisation simplifiées que l'on appellera wait et notify. Un processus léger peut décider de stopper son exécution en appelant la primitive wait. Son exécution ne sera pas reprise avant qu'un autre processus léger ne fasse un notify tid avec tid l'identifiant du premier processus léger. On définit l'état de cet ordonnanceur comme une paire de liste d'identifiants de processus légers selon qu'ils soient actifs ou en attente. On note que dans notre représentation, n'importe quel processus léger actif peut s'exécuter à tout moment : on est en présence d'un ordonnancement sémantique qui n'est pas déterministe. On définit donc l'état :

```
Definition sched := ((list thread_id) * (list thread_id))%type.
```

Pour simplifier, on établit une coercion permettant de nommer les appels systèmes :

```
Definition sys_coercion str: positive :=
match str with
| "creation" ⇒ 1%positive
| "wait" ⇒ 5%positive
| "notify" ⇒ 6%positive
| _ ⇒ 7%positive (* not handled sysevent *)
end.

Coercion sys_coercion :string >→ positive.
```

On définit l'état initial ne contenant qu'un processus léger actif :

```
Definition ext_init tid (sc: sched) := sc = ((tid :: nil), nil).
```

Les étapes possibles pour cet ordonnanceur sont définies de la façon suivante et attendue. On distingue six cas dans l'inductif que l'on traite séparément :

```
Inductive scheduler_step': sched' → SCevent → sched' → Prop :=
```

Pour le cas de création d'un processus léger, on doit vérifier que l'identifiant n'est pas déjà utilisé puis l'ajouter à la première liste.

```
| sys_create_step: ∀ sc l tid ntid
 (Hntid: ~ In ntid (fst sc) ∧ ~ In ntid (snd sc))
 (Hftid: In tid (fst sc)),
  scheduler_step' sc (SCsys "creation" tid (ntid :: nil) l) ((fst sc) ++ ntid :: nil, snd sc)
```

Le cas notify tid n'est exécutable que si l'élément tid se trouve dans la liste d'attente. Pour la valeur de tid, on récupère un ntid non borné à travers un Vptr car c'est la seule val qui autorise des éléments non bornés. On devrait utiliser Vint mais il semble qu'il faudrait ajouter des preuves de bornes et ce n'est pas le but ici. On utilise la fonction remove pour enlever un élément d'une liste :

```
| sys_notify_step: ∀ sc l tid ntid (Hftid: In tid (fst sc)) (Hntid: In ntid (snd sc))
 (Hval: l = (Vptr (Ptr (Zpos ntid) Int.zero)) :: nil),
 scheduler_step' sc (SCsys "notify" tid nil l) (ntid :: (fst sc), remove _ ntid (snd sc))
```

On n'ajoute pas de cas d'échec pour notify si l'élément n'est pas dans la liste car on veut rester le plus simple possible.

L'instruction de wait prends le processus léger actif et le place dans la liste des processus légers en attente.

```
| sys_wait_step: ∀ sc l tid (Hftid: In tid (fst sc)),
   scheduler_step' sc (SCsys "wait" tid nil l) (remove _ tid (fst sc), tid :: (snd sc))
```

Si tous les processus légers sont bloqués, l'ordonnanceur renvoie une erreur :

```
| sys_blocked_step: ∀ sc, (∀ tid, ~ In tid (fst sc)) → scheduler_step' sc SCfail sc
```

nstep traduit l'exécution d'un processus léger appartenant à la liste des processus légers actifs :

```
| nstep: ∀ sc tid (Hftid: In tid (fst sc)), scheduler_step' sc (SCstep tid) sc.
```

On définit decide_full de la façon la plus simple possible en ne propageant que les événements SCfail. Les seuls événements présents dans la trace de la machine globale sont donc des erreurs dans ce cas.

```
Definition decide_full (sc: sched) sev :=
match sev with
| SCfail ⇒ GMvisible Efail
| _ ⇒ GMtau
end.
```

On définit les processus légers actifs comme la concaténation des deux listes (cette définition de l'abstraction correspond en fait aux identifiants que l'on a déjà utilisé) :

```
Definition active_thread (sc: sched) := fst (snd sc) ++ snd (snd sc).
```

Les différentes propriétés de l'abstraction se démontrent de façon naturelle à partir de ces définitions simples. Par exemple, on démontre que tout processus léger vivant le restera toujours (max_thread_increasing) par une disjonction de cas sur scheduler_step.

6.2.3 Relation de simulation entre ordonnanceur

Dans cette section, on va montrer qu'une relation de simulation entre deux ordonnanceurs de la section précédente est possible. En considérant les ordonnanceurs simples et l'abstraction simplifié de relation entre ordonnanceur, on élimine la plupart des difficultés d'instances. Dans le cas le plus simple, on peut prendre l'égalité comme relation. On pose donc :

```
Definition sched_rel (scs sct: sched) := scs = sct.
```

On prouve facilement la notion de simulation arrière car deux états égaux peuvent exécuter la même étape pour se retrouver dans un même état. En effet,

en remplaçant sched_rel par l'égalité, on réduit valid_sched_bs a une propriété triviale :

```
valid_sched_bs_simpl: ∀ sched sched' ev,
 scheduler_step sched ev sched' →
 (∃ sched'', scheduler_step sched ev sched'' ∧  sched' = sched'')
```

Les relations d'initialisation et de blocages sont simplifiées de la même manière et se font très facilement en Coq grâce à la simplification de l'ordonnanceur que l'on décrit en section 6.2.1.

L'hypothèse sys_step sur la simulation d'événements systèmes se résout également très simplement, par une disjonction de cas.

Cette instance est très facile car on ne change rien au niveau de l'ordonnancement lors des phases de compilation. Cette méthode a été pensée pour permettre de prendre en compte des cas plus compliqués dans lesquels les ordonnanceurs source et cible sont différents. Par exemple, lors de la compilation d'un langage haut-niveau de squelettes algorithmiques, on va devoir compiler des instructions gérées globalement (impliquant plusieurs créations de processus légers, des communications, ...) vers des appels plus simples (wait/notify, lock). Ces appels haut-niveau existent pour l'ordonnanceur source mais pas pour l'ordonnanceur cible. Cela rend donc nécessaire une notion de relation entre simulateurs qui ne doivent pas être égaux. Néanmoins, ce manuscrit n'aborde pas les problèmes liés à la compilation de langage de squelettes et il est fort probable que les hypothèses définies à la section 4.5 nécessitent d'être reconsidérés.

Conclusion

Dans ce chapitre, on fournit des instances correspondant à des comportements réels pour le modèle mémoire et pour l'ordonnanceur ainsi que pour les relations respectives entre modèles source et cible. On fournit une instance d'un modèle séquentiellement consistant pour le modèle mémoire permettant une compréhension simple du comportement d'un langage et on fournit également une adaptation du modèle mémoire TSO à notre formalisme. Ces deux instances nous permettent d'espérer pouvoir instancier d'autres modèles mémoire faible à l'avenir. On instancie également l'ordonnanceur en fournissant une simplification de l'abstraction du modèle ainsi qu'une abstraction contenant des primitives de synchronisation et correspondant à un comportement réel.

Conclusion et perspectives

7

Sommaire

7.1 Bilan

En conclusion, nous avons accompli une première étape vers notre objectif initial qui est la formalisation de squelettes algorithmiques et l'écriture d'un compilateur pour ceux-ci dans un environnement complètement certifié. Nous sommes parvenus à formaliser de façon modulaire (en représentant les machines par des modules abstraits) des langages parallèles dont les sémantiques comportent des parties distinctes pour la gestion du modèle mémoire et de l'ordonnancement des processus légers. La modularité de ces formalisations nous a permis d'instancier les parties sémantiques par processus légers par des bases de langages déjà existant (tels que des versions de Cminor ou RTL que l'on trouve dans Compcert). Nous pouvons également utiliser deux modèles mémoires différents : TSO qui est une représentation du modèle utilisé par des processeurs multi-cœurs actuels, et un modèle séquentiellement consistant qui, à haut-niveau, permet une compréhension facile du comportement des programmes. On fournit également un modèle d'ordonnanceur abstrait gérant des primitives de synchronisation (wait, notify) que l'on pense facile à étendre à d'autres primitives.

On fournit un moyen de parler de compilation entre ces langages à travers les modules de relation entre modèles mémoire, ordonnanceurs, et langage par processus léger. En particulier, on réutilise les fonctions de compilation de CompcertTSO, et leur langages par processus léger que l'on prouve correct dans notre formalisme.

Enfin, la possibilité de gestion des primitives de synchronisation nous permet d'espérer pouvoir compiler un langage parallèle plus haut-niveau (tel qu'un langage de squelettes algorithmiques) vers notre langage source. Et cela peut se faire de façon relativement efficace car on peut utiliser des primitives systèmes (à travers l'ordonnanceur) : on n'est pas obligé de compiler toutes les primitives de synchronisation vers des attentes actives (read−modify−write). On note aussi qu'un tel langage, s'il respecte le formalisme que l'on donne, devient une preuve en un bloc d'une compilation d'un langage vers du code assembleur.

7.2 Perspectives

Ce travail offre de nombreuses perspectives : une première d'entre elles étant la preuve de correction d'une passe de compilation (adaptée de CompcertTSO) que nous ne sommes pas parvenus (faute de temps) à compléter avant la rédaction de ce mémoire.

La modularité de ce compilateur nous permet d'espérer pouvoir utiliser d'autres modèles mémoire faibles dans l'extension de compilateur (augmentant ainsi la portabilité du compilateur sur différentes machines). On pense également pouvoir utiliser différents ordonnanceurs qui permettraient d'ajouter, supprimer ou modifier les primitives de synchronisations.

Le formalisme proposé permet changer de modèle mémoire au cours de la compilation tout en restant correct (préservation de traces d'exécution des machines globales). Par exemple, on peut espérer prouver la correction d'un compilateur ayant un modèle mémoire séquentiellement consistant pour son langage source mais un modèle mémoire faible pour son langage cible.

Enfin, on pense que ce travail constitue une base pour un compilateur de langages plus haut niveau qu'il pourrait être intéressant de formaliser. La compilation correcte d'un tel langage peut nécessiter la définition de nouveaux outils permettant une simplification du travail de preuve de correction de compilation dans ce contexte (se basant sur la preuve de programme).

Annexes

Formalisation de l'article de McCarthy et Painter 1967

A

```
Require Import List. Import ListNotations.
Require Import Arith ZArith.

(** * Formalisation in Coq of "CORRECTNESS OF A COMPILER FOR
    ARITHMETIC EXPRESSIONS" by McCarthy and Painter, 1967 *)

(** ** The source language *)

Module Source.

 Definition variable : Set := nat.

 Inductive t : Set :=
 | var   : variable → t
 | const : Z → t
 | add   : t → t → t.

 Inductive In : variable → t → Prop :=
 | in_var : ∀ v, In v (var v)
 | in_add : ∀ v e1 e2, In v e1 ∨  In v e2 → In v (add e1 e2).

 Definition state := variable → Z.

 Definition c (x:nat) (s:state) := s x.

 Fixpoint value (e:t) (env:state) : Z :=
  match e with
    | var x ⇒ c x env
```

```
    | const v ⇒ v
    | add e1 e2 ⇒ ( (value e1 env) + (value e2 env) )%Z
  end.

End Source.

(** ** The target language *)

Module Target.

 (** Here an address is an option type so that the accumulator is the
    [None] value. *)

 Definition address : Set := option nat.

 Definition ac : address := None. (* The accumulator. *)

 Lemma eq_dec (l l' : address) : { l = l' } + { ~ l = l' }.
 Proof. repeat (decide equality). Qed.

 Inductive lt : address → address → Prop :=
 | lt_ac : ∀ l, lt ac (Some l)
 | lt_some : ∀ n1 n2, Peano.lt n1 n2 → lt (Some n1) (Some n2).

 Lemma lt_trans :
  ∀ a1 a2 a3,
   lt a1 a2 → lt a2 a3 → lt a1 a3.
 Proof.
  intros a1 a2 a3 H1 H2.
  case a1 as [a1 | ].
  - inversion H1 as [ | a1' a2' H1']. subst.
    inversion H2 as [ | a1' a3' H2']; subst.
    constructor. transitivity a2'; assumption.
  - inversion H1; inversion H2; subst;
    constructor.
 Qed.

 Inductive instruction : Set :=
 | li : Z → instruction
 | load : address → instruction
```

```
| store : address → instruction
| add : address → instruction.

Definition t := list instruction.

Definition state := address → Z.

Definition c (x:address) (η:state) := η x.

Definition a (l:address)(v:Z)(η:state) : state :=
 fun l' ⇒ if eq_dec l' l
        then v
        else η l'.

Lemma a_refl :
 ∀ x η y,  c y (a x (c x η) η) = c y η.
Proof.
 intros x η y.
 unfold a, c. simpl.
 destruct(eq_dec y x); subst; auto.
Qed.

Lemma a_a :
 ∀ z η x y vx vy ,
  c z (a x vx (a y vy η)) =
  if eq_dec x y then c z (a x vx η) else c z (a y vy (a x vx η)).
Proof.
 intros z η x y vx vy.
 unfold a, c. simpl.
 destruct(eq_dec z x) as [H|H]; auto.
 + destruct(eq_dec x y) as [H'|H']; auto.
   destruct(eq_dec z y)as [H''|H'']; auto.
   contradict H. congruence.
 + destruct(eq_dec z y)as [H'|H']; auto;
   destruct(eq_dec x y)as [H''|H'']; auto.
   contradict H. congruence.
Qed.

Lemma c_a :
 ∀ x η y v,  c x (a y v η) =
```

```
                if eq_dec x y then v else c x η.
Proof.
 intros x η y v.
 unfold a, c. simpl.
 destruct(eq_dec x y); auto.
Qed.

Fixpoint step (i:instruction)(η:state) : state :=
 match i with
  | li v     ⇒ a ac v η
  | load adr  ⇒ a ac  (c adr η) η
  | store adr ⇒ a adr (c ac η)  η
  | add adr   ⇒ a ac  ((c adr η) + (c ac η))%Z η
 end.

Fixpoint outcome (p : t) (η:state) : state :=
 match p with
  | [] ⇒ η
  | first::rest ⇒ outcome rest (step first η)
 end.

Lemma outcome_app:
 ∀ p1 p2 η,
 outcome (p1 ++ p2) η = outcome p2 (outcome p1 η).
Proof.
 induction p1.
 − trivial.
 − intros p2 η.
   simpl. rewrite  IHp1. trivial.
Qed.

Definition eqt (P:address→Prop)(η1 η2 : state) : Prop :=
 ∀ l, P l → c l η1 = c l η2.

End Target.

(** ** Compilation *)

Section compilation.
```

```
Variable loc : Source.variable → Target.address.

(** It is not explicit in the paper, but this property should
   hold: the accumulator cannot contain the value of one of
   the variables of the source expression. *)
Hypothesis loc_prop : ∀ x, loc x <> Target.ac.

Fixpoint compile (e : Source.t) (t : nat) : Target.t :=
 match e with
  | Source.const v ⇒ [ Target.li v ]
  | Source.var x   ⇒ [ Target.load (loc x) ]
  | Source.add e1 e2 ⇒
    (compile e1 t) ++ [Target.store (Some t)] ++
    (compile e2 (t+1)) ++ [Target.add (Some t)]
 end.

Definition valid (t:Target.address) := fun l ⇒ Target.lt l t.

Lemma valid_add_l:
 ∀ e1 e2 t x,
  (∀ x, Source.In x (Source.add e1 e2) → Target.lt (loc x) t) →
  Source.In x e1 → valid t (loc x).
Proof.
 intros e1 e2 t l H Hin. unfold valid in *.
 apply H. constructor. now left.
Qed.

Lemma valid_add_r:
 ∀ e1 e2 t x,
  (∀ x, Source.In x (Source.add e1 e2) → Target.lt (loc x) t) →
  Source.In x e2 → valid t (loc x).
Proof.
 intros e1 e2 t l H Hin. unfold valid in *.
 apply H. constructor. now right.
Qed.

Lemma valid_ac_some:
  ∀ t, valid (Some t) Target.ac.
Proof. intros. unfold valid, Target.ac; constructor. Qed.
```

```
Theorem compiler_correctness:
 ∀ (e:Source.t)(env:Source.state)(η:Target.state)(t:nat),
  (∀ x, Source.In x e → Target.lt (loc x) (Some t)) →
  (∀ x, Source.In x e → Source.c x env = Target.c (loc x) η) →
  Target.eqt (valid (Some t))
             (Target.outcome (compile e t) η)
             (Target.a Target.ac (Source.value e env) η).
Proof.
 induction e; intros env η t Ht Heq l Hl.
 − simpl. rewrite Heq; constructor.
 − trivial.
 − Opaque app. simpl. Transparent app.
  repeat rewrite Target.outcome_app.
  set(v1:=Source.value e1 env).
  set(v2:=Source.value e2 env).
  set(η1:=Target.outcome(compile e1 t) η).
  set(η2:=Target.outcome [Target.store (Some t)] η1).
  set(η3:=Target.outcome(compile e2 (t+1)) η2).
  assert(Target.eqt (valid (Some t)) η1 (Target.a Target.ac v1 η)) as Hη1.
  {
   intros address Haddress. unfold η1. simpl.
   rewrite IHe1 with (env:=env); trivial.
   − intros; eapply valid_add_l; eassumption.
   − intros; rewrite Heq; trivial.
     constructor. auto.
  }
  assert(Target.c Target.ac η1 = v1) as Hv1.
  {
   rewrite Hη1, Target.c_a by apply valid_ac_some.
   destruct(Target.eq_dec Target.ac Target.ac) as [H|H];
     firstorder.
  }
  assert(Target.eqt (fun l⇒Target.lt l (Some(t+1)) ∧ l<>Target.ac)
                    η2
                    (Target.a (Some t) v1 η)) as Hη2.
  {
   intros l' Hl'. unfold η2.
   simpl. rewrite Hv1.
   repeat rewrite Target.c_a.
   destruct(Target.eq_dec l' (Some t)) as [H|H].
```

```
  + trivial.
  + assert(Target.lt l' (Some t)).
   {
    destruct Hl' as [H1 H2].
    destruct l' as [ | n].
    − constructor.
     inversion H1 as [Ha' | n' n'' Hlt ]; subst.
     apply lt_le_S in Hlt. rewrite plus_comm in Hlt. simpl in Hlt.
     inversion Hlt; subst.
     + contradict H. trivial.
     + omega.
    − constructor.
   }
   rewrite Hη1 by auto.
   rewrite Target.c_a.
   destruct(Target.eq_dec l' Target.ac) as [H'|H']; subst.
   − destruct Hl' as [ _ Hl']; contradict Hl'; trivial.
   − trivial.
}
assert(Target.c (Some t) η2 = v1) as Hv1'.
{
  rewrite Hη2 by (split; [constructor; omega | intro; discriminate]).
  rewrite Target.c_a.
  destruct(Target.eq_dec (Some t) (Some t)) as [H|H].
  − trivial.
  − contradict H. trivial.
}
assert(Target.eqt (fun l⇒Target.lt l (Some(t+1)))
                  η3
                  (Target.a Target.ac v2 (Target.a (Some t) v1 η))) as Hη3.
{
  intros a H0.
  unfold η3.
  rewrite IHe2 with (env:=env).
  − rewrite Target.a_a.
   {
    destruct(Target.eq_dec Target.ac (Some t)) as [H|H].
    − contradict H. unfold Target.ac. intro. discriminate.
    − rewrite Target.a_a.
     destruct(Target.eq_dec (Some t) Target.ac) as [H'|H'].
```

```
        + contradict H'. unfold Target.ac. intro. discriminate.
        + do 2 rewrite Target.c_a.
          destruct(Target.eq_dec a Target.ac) as [H''|H''].
          * trivial.
          * apply Hη2. split; auto.
    }
  - intros x Hx.
    assert(Target.lt (loc x) (Some t)).
    apply Ht. constructor. now right.
    eapply Target.lt_trans.
    eassumption.
    constructor; firstorder.
  - intros x H.
    assert(Target.lt (loc x) (Some (t + 1)) ∧ loc x <> Target.ac).
    {
      split.
      - eapply Target.lt_trans.
        apply Ht. constructor; auto.
        constructor. omega.
      - apply loc_prop.
    }
    rewrite Hη2 by auto.
    rewrite Heq by (constructor; auto).
    rewrite Target.c_a.
    destruct(Target.eq_dec (loc x) (Some t)) as [Hloc | Hloc].
    + assert(Target.lt (loc x) (Some t)) by (apply Ht; constructor; auto).
      rewrite Hloc in H2; inversion H2; subst. omega.
    + trivial.
  - assumption.
}
simpl.
assert(Target.c Target.ac η3 = v2) as Hfinal2.
{
  rewrite Hη3 by constructor.
  rewrite Target.c_a.
  destruct(Target.eq_dec Target.ac Target.ac) as [H|H].
  - trivial.
  - contradict H; trivial.
}
assert(Target.c (Some t) η3 = v1) as Hfinal1.
```

```
    {
      rewrite Hη3 by (constructor; omega).
      do 2 rewrite Target.c_a.
      destruct(Target.eq_dec (Some t) Target.ac) as [Htac|Htac]; try discriminate.
      destruct(Target.eq_dec (Some t) (Some t)) as [Hteq|Hteq]; auto.
      now contradict Hteq.
    }
    rewrite Hfinal1, Hfinal2.
    do 2 rewrite Target.c_a.
    destruct(Target.eq_dec l Target.ac) as [Hlac | Hlac]; trivial.
    rewrite Hη3 by (eapply Target.lt_trans; [apply Hl| constructor; omega]).
    rewrite Target.c_a.
    destruct(Target.eq_dec l Target.ac) as [Hlac' | Hlac']; try contradiction.
    rewrite Target.c_a.
    destruct(Target.eq_dec l (Some t)) as [Hlsomet | Hlsomet].
    + rewrite Hlsomet in Hl. inversion Hl; subst; omega.
    + trivial.
  Qed.

End compilation.
```

Formalisation de la mémoire séquentielle

B

Sommaire

La mémoire est de type mem et contient les informations d'allocations et de valeur en mémoire pour tout emplacement mémoire arange définit comme par un pointeur et un offset.

Définition 25 (range) *Un arange est une adresse mémoire.*

```
Definition arange := (pointer * int)%type.
```

La bibliothèque sur la mémoire de Compcert fournit plusieurs prédicat permettant de retrouver l'informations contenus dans l'objet mem. Par exemple, range_allocated permet de savoir si un emplacement est alloué et load_ptr la valeur contenu à un emplacement donné.

Définition 26 (range_allocated)

```
Definition range_allocated (r : arange) (k : mobject_kind) (m : mem) : Prop.
```

range_allocated est vrai si le arange r est alloué avec le kind k dans la mémoire m.

Définition 27 (Load) *Renvoie la valeur situé à l'addresse p dans la mémoire m (la type de la donnée est c). Si la mémoire n'est pas alloué, load_ptr renvoit None.*

```
Definition load_ptr (chunk: memory_chunk) (m: mem) (p: pointer) : option val :=
  let (b, ofs) := p in load chunk m b (Int.unsigned ofs).
```

Ces types et définitions sont munis de notions de décidabilité et de différentiation élémentaire.

Définition 28 (décidabilité de l'égalité des arange)

```
range_eq_dec (x y : arange): {x = y} + {x <> y}.
```

Définition 29 (arange disjoint)

```
Definition ranges_disjoint (a b : arange) : Prop.
```

ranges_disjoint a b si deux arange ne se chevauchent pas.

Définition 30 (Décidabilité de la disjonction d'emplacement mémoire)

```
ranges_disjoint_dec:
 ∀ r r', {ranges_disjoint r r'} + {~ ranges_disjoint r r'}.
```

Définition 31 (Range inclus dans un range alloué)

```
Definition range_in_allocated (r : arange) (m : mem) : Prop :=
 ∃ r', ∃ k,
   range_inside r r' ∧  range_allocated r' k m.
```

Définition 32 (Décidabilité inclusion et allocation)

```
Lemma range_in_allocated_dec:
 ∀ (r : arange) (m : mem),
   {range_in_allocated r m} + {~ range_in_allocated r m}.
```

Définition 33 (Conversion en range) *Les accès aux tailles des arange de load_ptr se fait grace aux éléments de type memory_chunk représentant le type de la donnée en question.*

```
Definition range_of_chunk (p : pointer) (c : memory_chunk) : arange :=
 (p, Int.repr (size_chunk c)).
```

Opérations élémentaires sur la mémoire L'état de la mémoire est observable mais il est également possible de le changer grace à des opérations élémentaires. Dans Compcert, il y a 3 opérations possibles sur la mémoire: allocation, libération ou écriture. On va les utiliser via les fonctions alloc_ptr, store_ptr et free_ptr qui sont définies avec un type Option: elles retournent la mémoire après exécution d'une opération élémentaire ou None si cette opération est impossible.

Définition 34 (Alloc) *alloc_ptr renvoie la mémoire dans laquelle le arange r est alloué ou None si un problème d'allocation survient (emplacement déjà alloué, pointeur non aligné . . .).*

```
Definition alloc_ptr (r : arange) (k : mobject_kind) (m : mem) : option mem :=
 match tt with tt ⇒
   let '(Ptr b ofs, s) := r in alloc_mem (Int.unsigned ofs)
                               (Int.unsigned ofs + Int.unsigned s) k b m
end.
```

Définition 35 (Free) *Renvoie la mémoire dans laquelle l'emplacement vers lequel p pointe est libéré.*

```
Definition free_ptr (p : pointer) (k : mobject_kind) (m : mem) : option mem :=
  let (b, ofs) := p in free_mem (Int.unsigned ofs) k b m.
```

Définition 36 (Store) *Renvoie une nouvelle mémoire dans laquelle la valeur v est enregistré à l'emplacement pointé par p (et de taille/type chunk).*

```
Definition store_ptr (chunk: memory_chunk) (m: mem) (p: pointer) (v : val)
  : option mem :=
  let (b, ofs) := p in store_mem chunk (Int.unsigned ofs) v b m.
```

A ces opérations mémoires, des lemmes de bonne fondation sont ajoutés vis-à-vis de range_allocated et de load_ptr. Ces éléments sont l'objet de la section suivante.

Propriétes des opérations mémoires

Préservation de l'allocation après écriture :

```
store_preserves_allocated_ranges:
  ∀ (m : mem) (c : memory_chunk) (p : pointer) (v : val) (m' : mem),
  store_ptr c m p v = Some m' →
  ∀ (r : arange) (k : mobject_kind),
  range_allocated r k m <→ range_allocated r k m'
```

Mise à jour de l'emplacement écrit :

```
load_store_similar:
  ∀ (c c' : memory_chunk) (m : mem) (p : pointer) (v : val) (m' : mem),
  store_ptr c m p v = Some m' →
  size_chunk c = size_chunk c' →
  load_ptr c' m' p =
  Some (if compatible_chunks c c' then Val.load_result c' v else Vundef)
```

Préservation de la valeur d'emplacements disjoints :

```
load_store_other:
  ∀ (c : memory_chunk) (m : mem) (p : pointer)
    (v : val) (m' : mem) (c' : memory_chunk) (p' : pointer),
  store_ptr c m p v = Some m' →
  ranges_disjoint (range_of_chunk p c) (range_of_chunk p' c') →
  load_ptr c' m p' = load_ptr c' m' p'
```

Non préservation de la valeur d'emplacement se chevauchant :

```
load_store_overlap:
 ∀ (c c' : memory_chunk) (m : mem) (p p' : pointer)
  (v : val) (m' : mem),
 store_ptr c m p v = Some m' →
 ranges_overlap (range_of_chunk p c) (range_of_chunk p' c') →
 range_of_chunk p c <> range_of_chunk p' c' →
 chunk_allocated_and_aligned p' c' m' → load_ptr c' m' p' = Some Vundef
```

et

```
load_store_mismatch:
 ∀ (c c' : memory_chunk) (m : mem) (p p' : pointer)
  (v v' : val) (m' : mem),
 store_ptr c m p v = Some m' →
 load_ptr c' m p' = Some v' →
 ranges_overlap (range_of_chunk p c) (range_of_chunk p' c') →
 range_of_chunk p c <> range_of_chunk p' c' →
 load_ptr c' m' p' = Some Vundef
```

Allocation avant écriture :

```
store_allocated:
 ∀ (p : pointer) (c : memory_chunk) (v : val) (m m' : mem),
 store_ptr c m p v = Some m' →
 ∀ (r : arange) (k : mobject_kind),
 range_allocated r k m' <→ range_allocated r k m
```

Préservation des emplacements non alloués :

```
load_store_none:
 ∀ (p : pointer) (c : memory_chunk) (v : val)
  (m m' : mem) (c' : memory_chunk) (p' : pointer),
 store_ptr c m p v = Some m' →
 load_ptr c' m p' = None → load_ptr c' m' p' = None
```

Propriétés de la mémoires

Les range alloués sont disjoints :

```
Lemma ranges_are_disjoint:
 ∀ {m r1 k1 r2 k2}
  (RA1: range_allocated r1 k1 m)
  (RA2: range_allocated r2 k2 m),
```

```
r1 = r2 ∧ k1 = k2 ∨ ranges_disjoint r1 r2.
```

Bibliographie

[1] T. Abe and T. Maeda. Optimization of a general model checking framework for various memory consistency models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 14:1–14:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3247-7. doi: 10.1145/2676870.2676878. Cité page 41.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996. doi: 10.1109/2.546611. Cité pages 29 et 41.

[3] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of power and arm multiprocessor machine code. In L. Petersen and M. M. T. Chakravarty, editors, *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*, pages 13–24. ACM, 2009. doi: 10.1145/1481839.1481842. Cité page 29.

[4] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft Approach to Systems Verification. volume 5295 of *LNCS*, pages 209–224. Springer, 2008. doi: 10.1007/978-3-540-87873-5_18. Cité page 38.

[5] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. Cité page 38.

[6] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, pages 97–108, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596567. Cité pages 12 et 37.

[7] S. Berghofer and M. Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. *Electr. Notes Theor. Comput. Sci.*, 82(2):377–394, 2003. doi: 10.1016/S1571-0661(05)82598-8. Cité page 38.

[8] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *Programming Languages and Systems (ESOP)*, volume 8410 of *LNCS*, pages 107–127. Springer, 2014. doi: 10.1007/978-3-642-54833-8_7. Cité page 39.

[9] Y. Bertot. Coq in a hurry, 2006. `http://hal.inria.fr/inria-00001173`. Cité page 15.

[10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. doi: 10.1007/978-3-662-07964-5. Cité page 15.

[11] F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving*, volume 9236 of *LNCS*, pages 67–83. Springer International Publishing, 2015. doi: 10.1007/978-3-319-22102-1_5. Cité page 38.

[12] S. Blazy and X. Leroy. Formal verification of a memory model for C-like imperative languages. In *Formal Methods and Software Engineering (ICFEM)*, volume 3785 of *LNCS*, pages 280–299. Springer, 2005. doi: 10.1007/11576280_20. Cité page 38.

[13] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. doi: 10.1007/s10817-009-9148-3. Cité page 38.

[14] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, New York, NY, USA, 2008. ACM. doi: 10.1145/1375581.1375591. Cité page 29.

[15] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 107–115. IEEE Computer Society, 2013. doi: 10.1109/ARITH.2013.30. Cité page 39.

[16] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 392–403, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480930. Cité page 41.

[17] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hritcu, V. Sjöberg, and B. Yorgey. Software foundations. `http://www.cis.upenn.edu/~bcpierce/sf/current/index.html`, July 2014. version 3.1. Cité page 15.

[18] A. Chlipala. A verified compiler for an impure functional language. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 93–106. ACM, 2010. doi: 10.1145/1706299.1706312. Cité page 37.

[19] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2014. Cité page 15.

[20] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988. Cité page 15.

[21] T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88 International Conference on Computer Logic*, number 417 in LNCS, Tallinn, USSR, 1988. Springer. Cité page 15.

[22] M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003. doi: 10.1145/966221.966235. Cité page 37.

[23] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, École Normale Supérieure de Lyon, 1996. Cité page 15.

[24] J. Gosling, B. Joy, and G. Steele. The java language specification, 1996. Cité page 40.

[25] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In S. Drossopoulou, editor, *ESOP*, number 4960 in LNCS, pages 353–367. Springer, 2008. Cité page 41.

[26] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Cité page 14.

[27] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In H. Seidl, editor, *ESOP*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_20. Cité page 39.

[28] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811. Cité page 40.

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. Cité page 5.

[30] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Symposium on Principles of Programming Languages (POPL 2006)*, pages 42–54. ACM, 2006. doi: 10.1145/1111037.1111042. Cité page 38.

[31] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7): 107–115, 2009. doi: 10.1145/1538788.1538814. Cité pages 4 et 38.

[32] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. doi: 10.1007/s10817-009-9155-4. Cité page 44.

[33] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008. doi: 10.1007/s10817-008-9099-0. Cité page 38.

[34] A. Lochbihler. Verifying a Compiler for Java Threads. In A. D. Gordon, editor, *ESOP*, number 6012 in LNCS, pages 427–447. Springer, 2010. doi: 10.1007/978-3-642-11957-6_23. Cité page 39.

[35] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–391, New York, NY, USA, 2005. ACM. doi: 10.1145/1040305.1040336. Cité page 29.

[36] J. McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory*, volume 5 of *Proceedings of the Symposium in Pure Math*, pages 219–228, Providence, 1962. AMS. Cité page 37.

[37] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposium in Applied Mathematics, vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967. Cité pages 9, 10, 15 et 37.

[38] R. Milner and R. Weyrauch. Proving compiler correctness in a mechanized logic. In *7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–70, Edinburgh, UK, 1972. Edinburgh University Press. Cité page 37.

[39] F. Z. Nardelli, P. Sewell, J. Sevcik, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave. Relaxed memory models must be rigorous. *(EC)2*, 2009. Cité page 40.

[40] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010. doi: 10.1007/978-3-642-14107-2_23. Cité page 29.

[41] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27. Cité pages 5, 29 et 40.

[42] J. C. Process. Jsr-133: Java memory model and thread specification, Aug. 2004. URL `http://www.cs.umd.edu~{}pugh/java/memoryModel/jsr133.pdf`. Cité page 41.

[43] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000. doi: 10.1002/1096-9128(200005)12:6<445::AID-CPE484>3.0.CO;2-A. Cité page 40.

[44] S. Rideau and X. Leroy. Validating register allocation and spilling. In R. Gupta, editor, *CC*, volume 6011 of *LNCS*, pages 224–243. Springer, 2010. doi: 10.1007/978-3-642-11970-5. Cité page 39.

[45] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPoPP*, pages 161–172. ACM, 2007. doi: 10.1145/1229428.1229469. Cité page 29.

[46] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 379–391, New York, NY, USA, 2009. ACM. doi: 10.1145/1480881.1480929. Cité page 29.

[47] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-Memory Concurrency and Verified Compilation. In *POPL*, pages 43–54. ACM, 2011. doi: 10.1145/1925844.1926393. Cité page 39.

[48] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*, 60(3):22, 2013. doi: 10.1145/2487241.2487248. Cité page 39.

[49] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi: 10.1145/1785414.1785443. Cité page 29.

[50] M. Sozeau and N. Tabareau. Universe polymorphism in Coq. In *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 499–514. Springer, 2014. doi: 10.1007/978-3-319-08970-6_32. Cité page 17.

[51] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. Cité page 14.

[52] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, pages 17–27. ACM, 2008. doi: 10.1145/1328897.1328444. Cité page 39.

[53] J.-B. Tristan and X. Leroy. Verified validation of Lazy Code Motion. In *PLDI*, pages 316–326. ACM, 2009. doi: 10.1145/1542476.1542512. Cité page 39.

[54] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL*, pages 83–92. ACM, 2010. doi: 10.1145/1706299.1706311. Cité page 39.

[55] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *OOPSLA*, pages 691–707. ACM, 2014. doi: 10.1145/2660193.2660243. Cité page 30.

[56] V. Vafeiadis and F. Z. Nardelli. Verifying fence elimination optimisations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 146–162, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23701-0. Cité page 39.

[57] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Umm: An operational memory model specification framework with integrated model checking capability: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487, Apr. 2005. ISSN 1532-0626. doi: 10.1002/cpe.v17:5/6. Cité page 41.

Ce document a été écrit à l'aide l'éditeur de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

Sylvain Dailler

Extension paramétrée de compilateur certifié pour la programmation parallèle

Les applications informatiques sont de plus en plus présentes dans nos vies. Pour les applications critiques (médecine, transport, ...), les conséquences d'une erreur informatique ont un coût inacceptable, que ce soit sur le plan humain ou financier. Une des méthodes pour éviter la présence d'erreurs dans les programmes est la vérification déductive. Celle-ci s'applique à des programmes écrits dans des langages de haut-niveau transformés, par des compilateurs, en programmes écrits en langage machine. Les compilateurs doivent être corrects pour ne pas propager d'erreurs au langage machine. Depuis 2005, les processeurs multi-cœurs se sont répandus dans l'ensemble des systèmes informatiques. Ces architectures nécessitent des compilateurs et des preuves de correction adaptées.

Notre contribution est l'extension modulaire d'un compilateur vérifié pour un langage parallèle ciblant des architectures parallèles multi-cœurs. Les spécifications des langages (et leurs sémantiques opérationnelles) présents aux divers niveaux du compilateur ainsi que les preuves de la correction du compilateur sont paramétrées par des modules spécifiant des éléments de parallélisme tels qu'un modèle mémoire faible et des notions de synchronisation et d'ordonnancement entre processus légers. Ce travail ouvre la voie à la conception d'un compilateur certifié pour des langages parallèles de haut-niveau tels que les langages à squelettes algorithmiques.

Mots-clés : Compilation, vérification, parallélisme, modularité, assistants de preuve.

Parameterised extension of certified compiler for parallel programming

Nowadays, we are using an increasing number of computer applications. Errors in critical applications (medicine, transport, ...) may carry serious health or financial issues. Avoiding errors in programs is a challenge and may be achieved by deductive verification. Deductive verification applies to program written in a high-level languages, which are transformed into machine language by compilers. These compilers must be correct to ensure the non-propagation of errors to machine code. Since 2005, multicore processors have spread in all electronic devices. So, these architectures need adapted compilers and proofs of correctness.

Our work is the modular extension of a verified compiler for parallel languages targeting multicore architectures. Specifications of these languages (and their operational semantics) needed at all levels of the compiler and proofs of correctness of this compiler are parameterised by modules specifying elements of parallelism such as a relaxed memory model and notions of synchronization and scheduling between threads. This work is the first step in the conception of a certified compiler for high-level parallel languages such as algorithmic skeletons.

Keywords : Compilation, verification, parallelism, modularity, proof assistants.

LIFO - Batiment IIIA Rue Léonard de Vinci B.P. 6759 F-45067 ORLEANS Cedex 2